ANL-7818

CREATION OF HIERARCHIC TEXT WITH A COMPUTER DISPLAY

Wilfred J. Hansen

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy $3.00; Microfiche $0.95

ANL-7818
Mathematics and
Computers

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

CREATION OF HIERARCHIC TEXT WITH A COMPUTER DISPLAY

by

Wilfred J. Hansen

Applied Mathematics Division

June 1971

A Dissertation Submitted to the
Department of Computer Science
and the Committee of Graduate Studies
of Stanford University
for the Degree of
Doctor of Philosophy

Preface

The work reported in this thesis originally began with a plan for a flow chart language to create programs at a computer driven display console. The user would select boxes and draw their interconnections. Any box could be chosen from a set of primitives, or could be defined by a lower level flow chart. As I gained experience first with Algol and then other structured languages, I began to realize that flow charts were unnecessary. The nested, or *hierarchical*, structure of the program could be expressed in the typographical structure of the program text. With well chosen comments, appropriate indentation, and no labels, I found that I could write programs that were easily read because the flow of control proceeded in a straightforward way from the first line of text to the last.

Structured programs are possible in ALGOL because statements can be grouped by BEGIN-END 'parentheses'. As a result, a single conditional statement can include a number of subordinate statements whose execution depends on the condition. These subordinate statements can include further conditional statements, so that the program has a hierarchical structure. ALGOL is also hierarchic in the sense of levels of structural units: programs include procedures include statements include expressions include terms and so on. Both of these types of hierarchy are emphasized in the ALGOL defining document by the use of Backus-Naur Form (BNF) to describe the syntax.

Because of the simplicity and unity provided by BNF, I decided to base a program text construction and manipulation system on *generation of a program by applying syntactic rules*. A text includes certain

replaceable symbols (called non-terminal symbols in the BNF notation). For each of these the user selects a replacement from a set of choices displayed by the system. These choices are generated from a syntactic description of the programming language. Thus, rather than typing strings of characters, the user creates a program by selection. The entire system described below is based on the consequences and implications of this simple idea.

The first chapter is entirely introductory: the concept of *syntax* is not introduced until Chapter 2. Readers familiar with this concept may wish to skim until they reach the section Emily Text Structure in the second chapter. To assist such skimming, most of the introductory information has been summarized in the illustrations and their captions.

I presented the Emily system at the Computer Graphics 1970 Conference (Hansen, 1970). That paper described the basic system and proposed many advanced facilities that have now been implemented and are described in Chapter 3. It also outlined plans for measuring how useful the system was to the user. Unfortunately, there has not been time to train a user community, let alone make measurements. The details of Emily will be contained in three forthcoming reports: *Emily User's Manual*, *Emily Syntax Designer's Manual*, and *Emily System Documentation*.

The ideas in this thesis have a number of sources. Professor Niklaus Wirth introduced me to labelless programming and first expressed the concerns that led to the hierarchical hypothesis. My first exposure to graphical text editing was the excellent TVEDIT system designed

by Brian L. Tolliver. I have learned a great deal from the work of Dr. Douglas Engelbart. Above all, I am indebted to my advisors, Professors John C. Reynolds and William F. Miller. By understanding the implications of my explanations, Professor Reynolds sometimes knew more about Emily than I did. Professor Miller provided continuous encouragement; without his support the Emily system might never have been planned.

A route of evanescence
With a revolving wheel;
A resonance of emerald,
A rush of cochineal;

And every blossom on the bush
Adjusts its tumbled head, -
The mail from Tunis, probably,
An easy morning's ride.

Emily Dickinson

TABLE OF CONTENTS

LIST OF FIGURES

CREATION OF HIERARCHIC TEXT WITH A COMPUTER DISPLAY

by

Wilfred J. Hansen

Abstract

Paper and pencil, the traditional tools for creation of computer programs, assist the programmer very little. Proper punctuation demands precision, review of existing text requires clumsy paper shuffling, text modification is difficult and messy. In conjunction with a file storage device and a graphic display unit, a computer can provide a more flexible medium, but early systems still treated the text as an unstructured string of characters. Emily, the system described in this paper, avoids these problems because text is created, viewed, and modified in terms of the structure imposed by the syntax of the programming language.

To describe languages for the Emily system, a syntactic formalism was developed. Based on Backus-Naur Form, this formalism can describe identifier block structure, indentation, and conditional display of text. The user creates text by selecting among choices displayed by the system under guidance of a language description in this formalism.

The interface between man and system was designed in accordance with a set of user engineering principles. Thirteen principles are discussed under the headings of 'minimize memorization', 'optimize operations', and 'engineer for errors'.

Results of a rudimentary comparison with a system for unstructured strings show that the user took slightly longer with Emily, but made fewer mistakes.

1. Introduction

Good communication is vital in this age of rapid change. Man-man communication is essential for basic human understanding; man-machine communication is necessary to control our technology. Technology, in turn, has contributed to better communication by providing more effective intermediaries. Among these intermediaries are a number of computer systems that help a user build and modify files of text. As diagrammed in Figure 1.1, the user sits at a console and the file of text is stored in one of the peripheral devices attached to the computer. Later, the text is read and acted upon:

another user may read the text via the same interactive system,

the computer may be directed to read the text and act on the instructions therein, or

the creator of the text may read it and revise it.

This thesis describes an experimental text manipulation program. Called Emily, this program is used to create and manipulate texts that can be described by *formalized languages*.

A language is formalized if the set of valid strings in that language is described by a notational mechanism sufficiently precise to determine whether any given string is indeed written in the language. The main consideration in this paper will be computer programming languages that can be described by the Backus-Naur Form (BNF) notation (Backus, 1959). While a text is being constructed according to the rules of this notation, it contains certain replaceable symbols where the text is incomplete. The notational definition of the language specifies a

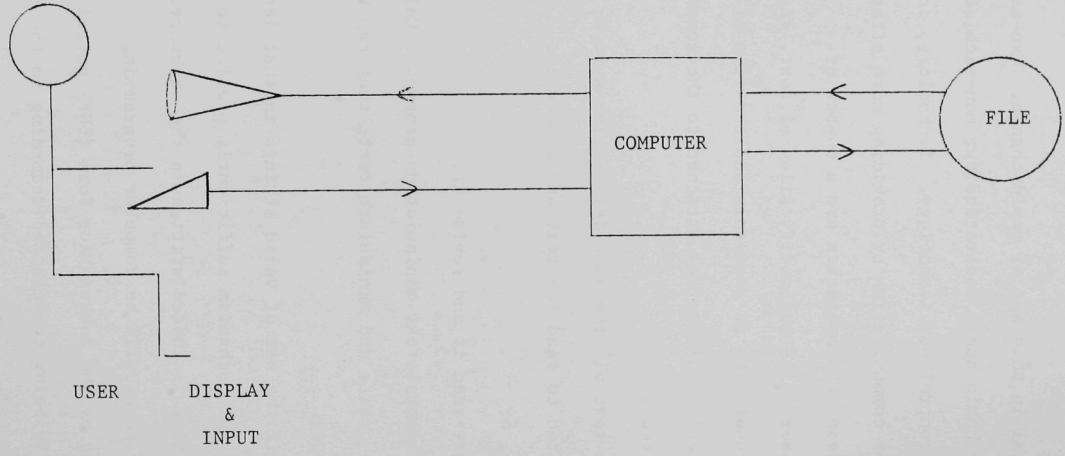


Figure 1.1. Path Between User and File

The user is working with information stored on the file. He views a portion of the text on the display and gives instructions with two keyboards and a light pen.

relatively small set of possible replacements for each of these symbols. Each replacement may itself include one or more of these replaceable symbols, but eventually all must be replaced by completed text. For example, the symbol <STMT> might be replaced by

```
DO <ARITHV> = <ARITHX> TO <ARITHX>;
   <STMT*>
END;
```

where the symbols in brackets are other replaceable symbols.

The key distinction between Emily and other text creation systems is the manner in which the user creates text. With other systems, the user types the text as a string of characters. With Emily, the user selects replacements. For each replaceable symbol, Emily displays a list of the valid replacements and the user selects the one that builds his text in the desired direction. One advantage of this approach is that the user is prevented from making typographical errors like omitted commas or unbalanced parentheses. However, the major advantage is that the resulting text is *hierarchical*.

The Hierarchical Hypothesis

A hierarchy is a collection of objects organized in levels so that each object (other than the topmost) is immediately subordinate to exactly one object and all objects are superordinate to zero or more other objects. In general, subordinate objects need not be ordered, but the Emily system always displays hierarchies in a specific order. In Emily, the 'objects' are pieces of text. The rules of the formalized language specify an organization such that a piece of text is

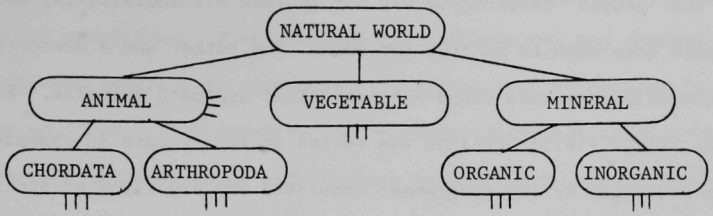
superordinate to the several pieces of text of which it is composed. For example, the text of a procedure contains the text of individual statements.

Hierarchies are an important aid to thought because they are easily visualized and help to localize analysis. Among the many possible visualizations of hierarchies are those shown in Figure 1.2. Each of these diagrams could be continued to any level of nesting, but in practice this can become unwieldy. Instead, the diagram is broken off at some level and nodes at that level are expanded further in separate diagrams. In an ideal hierarchy, there are no interactions among the nodes subordinate to a given node. Consequently, the structure can be studied piecemeal by studying each node and its immediate subnodes. The name or other identification on each subnode should specify its contents, so when a subnode is studied the reader need only verify that the subnode lives up to its name. Thereafter when he encounters a name, the reader need not reexamine the corresponding node. Even though few hierarchies are ideally non-interactive, the imposition of hierarchical structure is an aid to study and understanding. Once the exceptions are explained, the bulk of the information can readily be examined in its hierarchical structure.

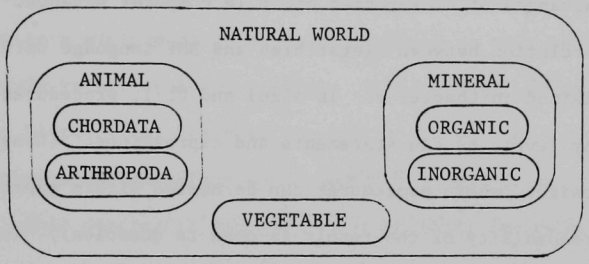
Central to the Emily project has been an assumption that can be called the 'hierarchical hypothesis:'

> People think in terms of hierarchies and systems that manipulate hierarchies are better suited to creative work than systems that treat information as unstructured text.

a) tree



b) nested areas



c) nested line segments

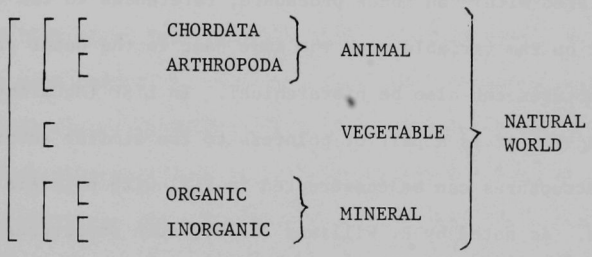


Figure 1.2. Visualizations of a Hierarchy

Each of these mental images of a hierarchy is convenient for a different purpose. The Emily system displays hierarchies with the representation shown in the next figure.

Though the 'people' referred to may not include all individuals, this hypothesis does seem to be true for some. The author and a number of other investigators have found a hierarchical approach valuable. For instance, Wirth (1971) outlines the series of refinements an example program undergoes as the programmer considers the algorithm at successively deeper levels.

Many instances of hierarchies occur in computer science. The fundamental relation between hierarchies and BNF language definition will be explained in Chapter 2. In Algol and PL/I, procedures can be nested to any level, as can statements and expressions. (Languages have even been designed where procedures can be nested within expressions, though the readability of the result is open to question.) The nesting of procedures controls the influence of declarations. If an identifier is redeclared within an inner procedure, references to the variable have no effect on the variable with the same name in the outer procedure. Data structures can also be hierarchical. In LISP (McCarthy, 1962), each data element is a pair of pointers to two similar subordinate elements. Similar structures can be constructed in PL/I with pointers and based variables. As noted by R. Williams (1970), such structures are easier to build and maintain than arbitrary collections of nodes connected at random.

In addition to structural hierarchies, computer programs often exhibit logical hierarchies. Sometimes referred to as 'modular programming', logical hierarchies are a style of programming where each process is defined in terms of a number of primitive processes at a

lower level. These primitives are then defined in terms of a still lower set of primitives. This division into hierarchies has several advantages. Most modifications to system specifications require changes in only a few routines. If a primitive has been dutifully called upon wherever its function is required, a modification to that primitive can modify the behavior of the entire system. Moreover, testing of hierarchic systems is additive rather than multiplicative. Interactions between modules are kept to a minimum so it is only necessary to test each path through each module rather than all possible paths through the system.

The hierarchical hypothesis seems to explain the power of the label-less programming discipline espoused by E. W. Dijkstra (1968a, b) and others. This discipline requires the programmer to eschew labels and GOTO's and control the flow of program execution with appropriate use of IF-THEN-ELSE and DO-WHILE. (A theorem by Böhm and Jacopini (1966) guarantees that these two statement types are adequate.) Further, the programmer must indent his text correctly so that text subordinate to a conditional clause is indented from the statement containing that clause. Such indentation is readily accomplished due to the absence of labels, otherwise the subordination of text might not be as clear. The rewards of this discipline are that the program is more likely correct because conditions must be considered carefully and the program text is more readable because the flow of control at execution time is mirrored in the typographical structure of the text. With appropriate comments, the text is sufficiently readable to supplant the traditional 'flow' chart. The general flow is represented by the least indented lines, while detailed flow is shown by the lines in between.

The scope of the Emily project was to build a hierarchical text system and to demonstrate the possibilities of this approach. A complete test of the hypothesis would require extensive testing on a large number of users over a period of time. However, implementation of Emily has shown that such experimentation is justified because in several areas the system has significant advantages over other text editing systems.

Other Text Manipulation Systems

Many systems have been built for entering unstructured text into a computer. Among the best of these are TVEDIT (McCarthy, 1967), QED (Thompson, 1968), and Wylbur (Stanford, 1968). Further examples are described in (Rice, 1970). Two interesting systems that prevent errors in the created program are described in (Bratman, 1968) and (Cameron, 1967). James E. George has implemented two special purpose systems that permit text creation by syntax controlled substitution (1967, 1968). These systems were influential in the author's design of Emily.

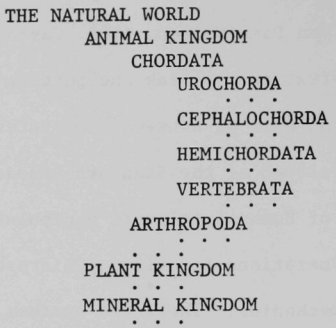
In a visionary 1945 article in Atlantic Monthly, Vannevar Bush described what he called a Memex (Bush, 1945). This device would replace the desk as a scholar's workspace. Displays would present articles he had entered in his files and a keyboard would let him enter his own thoughts. Essential to the Memex was the provision for entering notes and references at any point in a file. While reading, references could be examined by pressing a button.

Two systems have been inspired by Bush's article. The Hypertext system by Nelson and van Dam (Nelson, 1965; Carmody, 1968) permits the user to annotate his text and to link one portion of text to another. The text appears to the user as a network of interconnected blocks. The other system, developed at the Stanford Research Institute's Center for the Augmentation of Human Intellect, manipulates hierarchical text (Engelbart, 1968). Operations for viewing hierarchies are an integral part of the display mechanism. These two systems, in turn, provided inspiration for the Emily system. But with these systems, the user must type each character of his text; with Emily, the user creates text by selecting among options displayed by the system. Furthermore, the hierarchies created with Emily are not usually general hierarchies like those in Engelbart's system; instead they are hierarchies with various types of components depending on (and limited by) the language in which the text is being created.

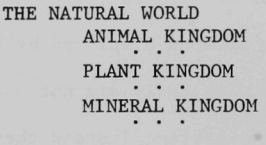
The Appearance of Emily Text

Once hierarchical text has been created with Emily, it can be viewed with four operations--contraction, expansion, descent, and ascent. As an illustration, several views of a typical hierarchy are shown in Figure 1.3. Three dots represent one or more sub-hierarchies that are present in the full hierarchy but not displayed in a particular view. In 1.3b, some of the structure has been *contracted* into three dots. In 1.3c, the top of the view has *descended* into the structure. In this latter view, the sub-hierarchies of UROCHORDA have been *expanded* one

a) One view of a typical hierarchy.



b) View after contracting the sub-hierarchies of ANIMAL KINGDOM.



c) View after descent into a sub-hierarchy.

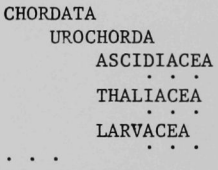


Figure 1.3. Three Views of a Typical Hierarchy

Three dots represent one or more sub-hierarchies not visible in a view. In (c) the last three dots represent the ARTHROPODA and other sub-hierarchies of the animal kingdom. The three just above represent the sub-hierarchies under LARVACEA. There are only three orders under UROCHORDA so there are no dots for further sub-hierarchies.

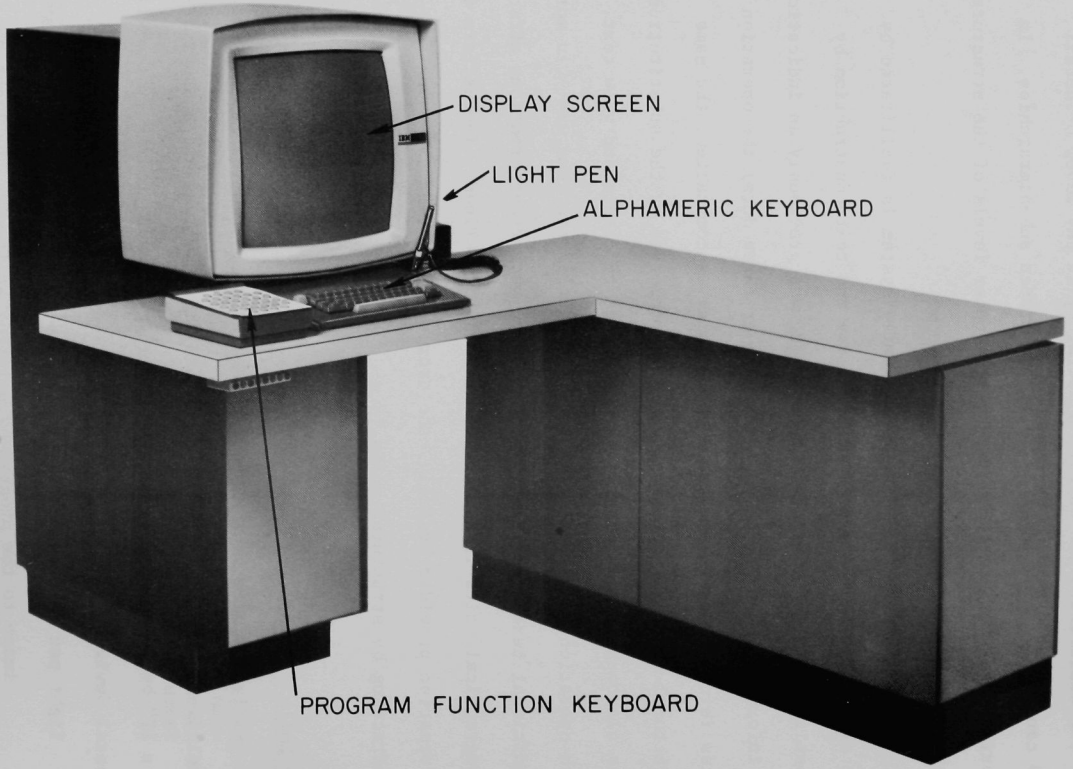
level. The user of a system offering these powerful operations for viewing hierarchies can observe a hierarchy at any level of detail and can easily study the relationships between sub-hierarchies. In a printed form of such a hierarchy, the upper levels of the structure might be separated by many pages.

The user's understanding of the Emily system is facilitated by the similarity of the expansion operation and text construction by substitution. In both cases, where there was previously an indication of information (undefined sub-hierarchy or three dots) the operation makes information appear. When the display is contracted, the same structural unit can be contracted as was inserted by the substitution. Thus although there are separate operations to create and view text, both operations appear to transform the text in the same way. In more traditional text handling systems the user sees transformations affect typographical units like characters, words, and lines. In the form of information provided by Emily, the user sees the text expanding and contracting by structural units.

The Emily System

Emily has been implemented for an IBM 2250 Graphic Display Unit, model 3. The 2250--see Figure 1.4--can display lines and characters on a 12" by 12" screen. The user can give commands to the system with several devices:

light pen - The user indicates a particular part of the display image to the program by pointing at it with the light pen.



Courtesy: IBM Corporation

Figure 1.4. IBM 2250 Graphic Display Unit

program function keyboard - There are thirty-two buttons. The meaning of a button push is controlled by Emily.

alphanumeric keyboard - This is a standard typewriter keyboard, but typed characters are entered in the display. The computer only responds to entered information when a special key is pushed.

At Argonne National Laboratory, where Emily has been implemented, the 2250 is attached to an IBM 360 model 75. The 75 is under control of the MVT version of OS/360. Unit record input/output is controlled by ASP in an attached 360/50. The 360/75 has one million bytes of main core and one million bytes of a Large Capacity Storage Unit.

The Emily system itself requires 60K bytes of main core (the maximum permitted for a 2250 job at Argonne) and about 400K bytes of LCS. Emily is written in PL/I (IBMUK, 1968) and uses the Graphic Subroutine Package (IBM, 1968) to communicate with the 2250. Files for Emily are stored on a 2314 disk pack. Emily is table driven and can manipulate text in a large variety of formal languages. To date, tables have been created for four languages: PL/I, GEDANKEN (Reynolds, 1970), a simple hierarchy language for writing thesis outlines, and a language for creating syntax definitions.

The Emily system outlined above will be described in the second and third chapters. Chapter 4 details the Emily design philosophy. These 'User Engineering Principles' should govern the design of any interactive system. The user must be given power to solve his problem, but the operations available must be logically organized so they can be

readily remembered or deduced from simple principles. The fifth chapter describes the formalism used within Emily to describe languages. Though similar to BNF, this formalism includes codes to format the text and provide context dependent punctuation. Some observations concerning the hierarchical hypothesis and other experience with Emily are in Chapter 6.

2. Basic Text Creation and Display

Before describing how an Emily user creates text, this chapter will show that (1) a hierarchy can be described by a syntax, and (2) a syntax imposes a hierarchical structure on text. The computer applications of formal syntax began with the use of BNF to describe Algol (Backus, 1959). In most systems based on BNF, a syntax describes the language and a programmer writes a linear string of characters that is supposed to satisfy that description. The compiler for the language then dissects this string to determine how it matches the syntax. The resulting structural information enables the compiler to 'understand' the programmer's intentions. But if the string does not meet the syntax (missing comma, perhaps), the compiler must signal an error. The Emily system avoids such errors because the text is created according to the syntax from the start.

Hierarchical Structure and Syntax

A text representation of a hierarchy is constructed according to a simply defined pattern. For example, the structure of the hierarchy in Figure 1.3 can be defined as follows:

(1) A title is a string of characters.

(2) A section can be a single title.

(3) A section can also be a title followed by a sequence of sections indented from the title.

(4) A hierarchy is a section.

This pattern can be described by what is known as a *syntax*. Corresponding to the above pattern is the syntax shown in Figure 2.1. Symbols

```
<TITLE> IS A STRING                                          (1)
<SECTION> ::= <TITLE>                                        (2)
            |<TITLE>{start indentation}<SECTION*>             (3)
<SECTION*> ::= {new line}<SECTION><SECTION*>                 (3a)
             |                                               (3b)
<HIERARCHY> ::= <SECTION>                                    (4)
```

Figure 2.1. Syntax Describing the Pattern Given in the Text

The numbers at the right correspond to rule numbers in the text. This syntax is the BNF description of the structure of the hierarchy in Figure 1.3.

that can be replaced by strings begin with '<', end with '>' and contain a name that usually has some relation to the meaning of the string generated by replacing the symbol. These symbols are called 'non-terminal' symbols because they must be replaced before the text is complete. Possible replacements are specified by the rules of the syntax: the strings following '::='. Alternate replacements for a non-terminal are separated by '|'. In 2.1, rule (1) specifies that the non-terminal <TITLE> must be replaced by a string of text. Rule (3) specifies one possible replacement for <SECTION>. After this replacement there are two non-terminals in the text instead of one. Rule (3a) shows how to generate a list of <SECTION>'s and (3b) shows how to end the list. The effect of the latter rule is to delete <SECTION*> from the text.

To be more precise, a BNF definition for a formal language has three parts--a set of *terminal symbols*, a set of *non-terminal symbols*, and a set of *syntactic rules*. The terminal symbols are those characters and strings of characters (punctuation, reserved words, identifiers, constants) that can be part of the completed text. The non-terminal symbols are a specific set of symbols introduced only to help describe the structure of the formal language. Every non-terminal symbol must be replaced by terminal symbols before the entire text is complete, but the only allowable replacements for a given non-terminal are specified by the syntactic rules. In each rule, the given non-terminal is on the left followed by '::=' followed by the sequence of symbols that may replace the non-terminal. As an example, Figure 2.2 shows a portion of

```
1          <STMT> ::= DO <ARITHV> = <ARITHX> TO <ARITHX>;
                          <STMT*> END;

2                   | <ASGN STMT>

3         <STMT*> ::= <STMT>

4      <ASGN STMT> ::= <ARITH> = <ARITHX>;

5         <ARITHX> ::= <ARITH>

6                   | <ARITHV>

7                   | <NUMBER>

8                   | <ARITHX> + <ARITHX>

9       <ARITHX*> ::= <ARITHX>

10        <ARITHV> ::= <ARITH>

11                  | <ARITH> (<ARITHX*>)

12        <ARITH> IS AN IDENTIFIER

13        <NUMBER> IS A CONSTANT
```

Figure 2.2. Portion of Syntax for PL/I

Each rule specifies a possible replacement for the non-terminal on the left. Rules 12 and 13 specify special classes of terminal symbols; the user can enter replacements for these symbols from the keyboard.

the syntax for PL/I. Figure 2.3 shows the steps in the generation of a DO loop according to this syntax.

Note now that a string generated according to a syntax is not simply a sequence of characters, but can be divided into hierarchies of substrings on the basis of the syntactic rules. Each non-terminal in the sequence of symbols for a rule generates a sub-sequence. The DO statement in Figure 2.3 can be one of a sequence of statements in some higher DO loop and can also contain a subordinate sequence of statements (generated by <STMT*>). A convenient visualization of the hierarchy imposed by a syntax is the tree representation in Figure 1.2a. Replacement of a non-terminal by a rule can be thought of as replacing the non-terminal with a pointer to a copy of the rule. The structure for the string in Figure 2.3 can be diagrammed as shown in Figure 2.4. Each syntactic rule used in the generation of the string is represented by a *node* (a rectangle). The node contains one pointer to a subordinate node for each non-terminal in the syntactic rule. The subordinate node is called a subnode or a descendant, while the pointing node is called the parent.

Emily Text Structure

Text in the Emily system is stored in a *file*, which may contain any number of *fragments*. Each fragment has a name and contains a piece of text generated by some non-terminal symbol. Generated text is physically stored in a hierarchical structure like that in Figure 2.4. Each node is a section of memory containing (a) the number of the syntax rule for which this node was generated, and (b) one pointer to each subnode. In

```
<STMT>                                                    1

DO <ARITHV> = <ARITHX> TO <ARITHX>                        10
    <STMT*>
END;

DO <ARITH> = <ARITHX> TO <ARITHX>;                        12
    <STMT*>
END;

DO I = <ARITHX> TO <ARITHX>;                              7
    <STMT*>
END;

. . .                                          13,7,13,3,2

DO I = 1 TO 20;                                           4
    <ASGN STMT>
END;

. . .                                          12,8,5,12,6,11,
                                                12,9,5,12
DO I = 1 TO 20;
   S = S + A(I);
END;
```

Figure 2.3. Steps in the Generation of a DO loop

In each step, the non-terminal in the rectangle is replaced according to the rule whose number appears at the right.

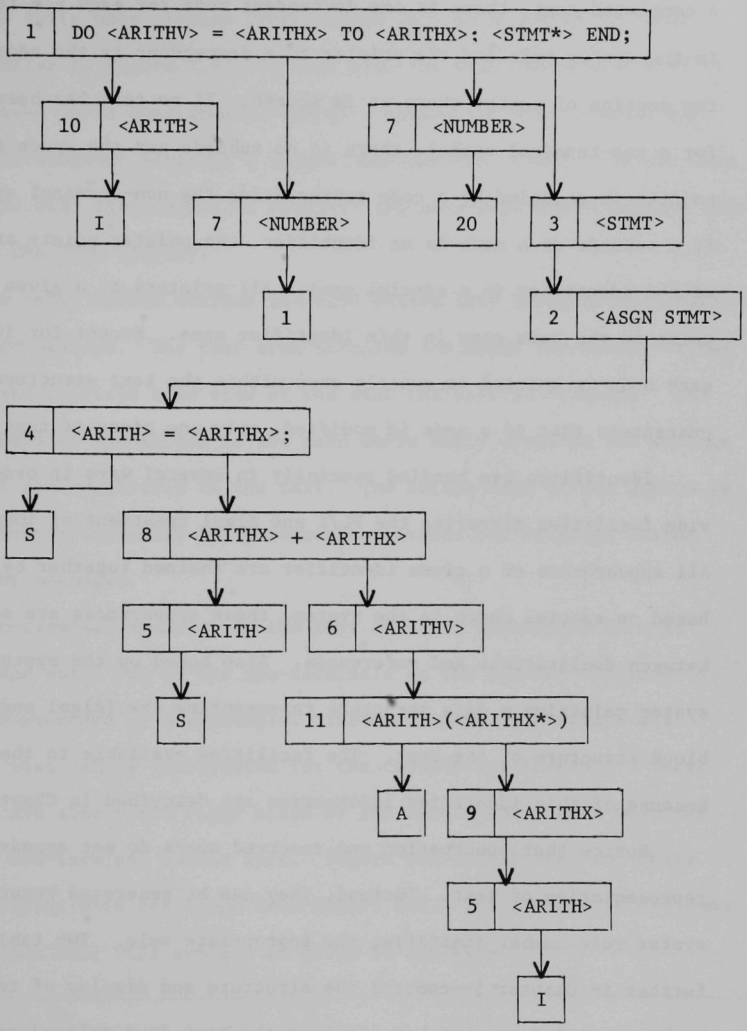


Figure 2.4. Hierarchical Structure Imposed by Syntax

This is a structure diagram of the DO loop generated in Figure 2.3. It is also a diagram of the Emily data structure to represent this text.

a completed text, there is one descendant node for each non-terminal in the syntax rule and the pointer to a descendant is the address of the section of memory where it is stored. If no text has been generated for a non-terminal symbol, there is no subnode and the space for the pointer is occupied by a code representing the non-terminal symbol. If a subnode of a node is an identifier, the pointer points at a copy of the identifier in a special area. All pointers at a given identifier point to the same copy in this identifier area. Except for identifiers, each node is pointed at exactly once within the text structure. This guarantees that if a node is modified, only one piece of text is affected.

Identifiers are handled specially in several ways in order to provide facilities mirroring the PL/I and Algol treatment of identifiers. All appearances of a given identifier are chained together by pointers. Based on special codes in the syntax, these appearances are separated between declarations and references. Also based on the syntax, the system maintains a data structure representing the (Algol and PL/I) block structure of the text. The facilities available to the user because of this identifier information are described in Chapter 3.

Notice that punctuation and reserved words do not appear in this representation of text. Instead, they can be generated because the syntax rule number identifies the appropriate rule. Two tables--described further in Chapter 5--control the structure and display of text. The important point to note here is that the text is displayed as a string of characters, even though it is stored internally as a hierarchical structure.

Creating Text

The Emily user creates hierarchical text in a series of steps very similar to Figure 2.3. In each step the right side of a rule is substituted for a non-terminal symbol. Before the user creates any text, the fragment contains a single non-terminal symbol. In the case of Figure 2.3, that symbol is <STMT>. The user sees the result of each step on the 2250 display.

The Emily system divides the 2250 screen into three areas: text, menu, and message. The text area occupies the upper two-thirds of the screen and displays some view of the text the user is creating. The lower third of the screen is the menu where Emily displays the strings the user can substitute in the text. The bottom line of the screen is the message area, where Emily requests operands and displays status and error messages.

Non-terminal symbols* in the text area are underlined to make them stand out. One of the non-terminals is the *current non-terminal* and is surrounded by a rectangle. The menu normally displays all strings that can be substituted for the current non-terminal. These strings are simply the right sides of the syntax rules that have the current non-terminal on the left. Figure 2.5 illustrates the Emily screen layout with the steps from Figure 2.3. The choices in the menu reflect the full PL/I syntax, as given in Appendix C.

---

*When it is displayed, a non-terminal is the end (or terminal) of a branch of the hierarchical structure. It is called a non-terminal because it must be replaced with a string of terminals before the text is complete.

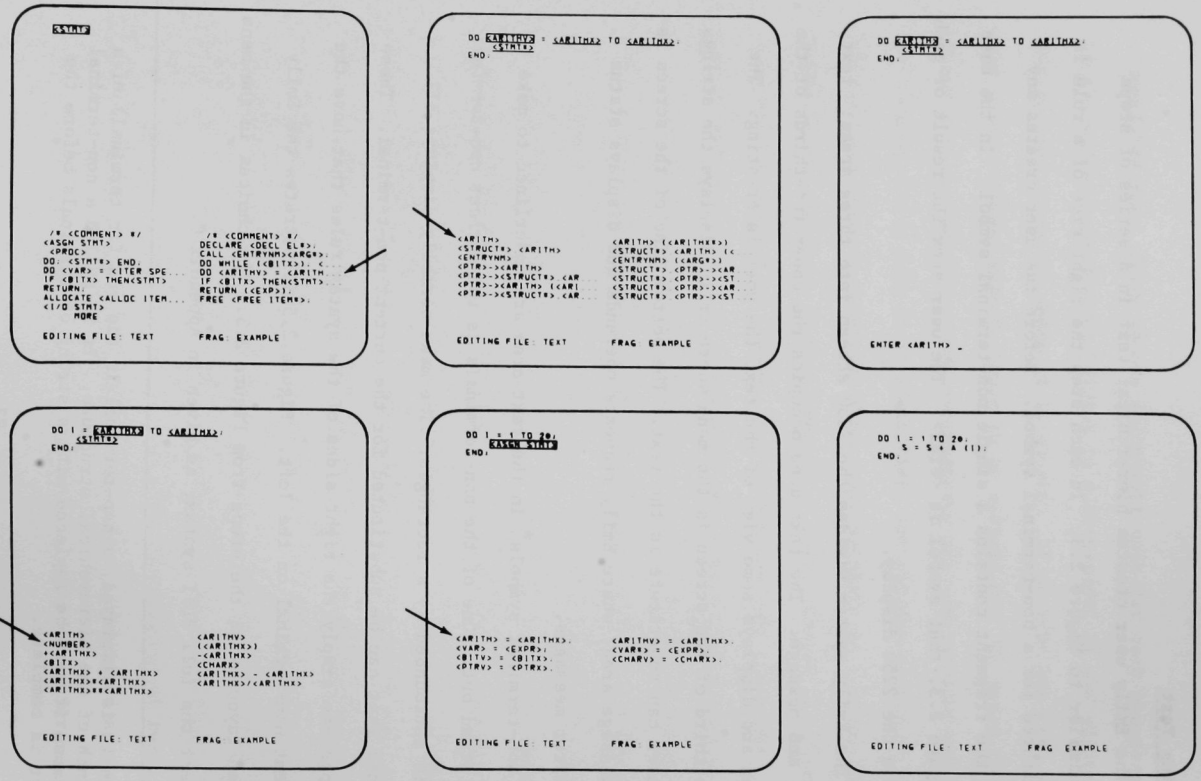


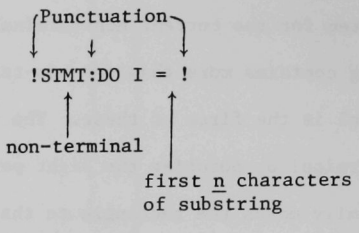
Figure 2.5. Generation of a DO Loop with Emily

These photographs show the same steps as shown in Figure 2.3. The menu displays all the choices available in the implemented PL/I syntax. An arrow indicates the syntax rule the user will select next. Up to twenty-two lines of text may be shown in the text area, so it appears empty with only 3 lines.

When the user points the light pen at an item in the menu Emily substitutes that item for the current non-terminal. Usually, the substitution string contains more than one non-terminal and the new current non-terminal is the first of these. The user can also change the current non-terminal by pointing the light pen at any non-terminal in the display. Emily moves the rectangle to that non-terminal and changes the menu accordingly. When the current non-terminal is an identifier, the menu displays identifiers previously entered in the required class (some of the classes for PL/I are <ARITH>, <CHAR>, and <ENTRYNM>). The user may select one of these, or he may enter a new identifier from the keyboard. Constants are also entered from the keyboard. There are no special provisions for comments. If they are to be allowed in the text, they must be described in the syntax as constants.

Viewing Text

In the first chapter, contracted portions of a hierarchy were represented by three dots (Figure 1.3). To assist the user, the Emily system represents contracted information with a symbol called a *holophrast*. This symbol begins and ends with an exclamation mark and contains two parts separated by a colon. The first part is the non-terminal symbol that generated the sub-structure and the second part is the first few characters of the contracted string. Figure 2.6 details this structure and shows examples of holophrasts. It is important to note that contraction to a holophrast only changes the view of the file, it does not modify the file itself. Moreover, the




```
DO I = 1 TO 20;
   !STMT:S = S +!
END
```

```
DO I = 1 TO 20;
   S = !ARITHX:S + A(I!;
END;
```

Figure 2.6. Examples of Holophrasts

All three examples show the DO loop, but each has been contracted differently. The user may change n, the number of characters of the substring. In the examples, n is seven.

user never types a holophrast; they are displayed only as a result of contraction in the hierarchy.

The user contracts a structural unit in the display by pushing a button on the program function keyboard and then pointing at some character in the text. A holophrast replaces that character and all other characters generated by the non-terminal that generated the selected character. For example, in the DO loop, the plus sign is generated by the node for rule 8. If the user contracts the text by pointing at that plus sign, the result is the third example in Figure 2.6. When text is contracted by pointing at a holophrast, the father of the indicated node contracts to a new holophrast. Text characters are the only characters displayed in the second part of this new holophrast. No holophrast ever contains the exclamation marks or other parts of a subordinate holophrast. To expand a holophrast back to a string, the user returns to normal text construction mode and points the light pen at the holophrast.

The operations to ascend and descend in the text hierarchy are also invoked by program function buttons. To descend in the hierarchy the user pushes the IN button and points at a part of the text. The selected node becomes the new *display generating node*; subsequent displays show only this node and its subnodes. The OUT button lets the user choose among the ancestors of the display generating node and then makes the selected ancestor the new display generator.

3. Additional Facilities

The simple system described in the preceding chapter was implemented and illustrated the concepts of hierarchy viewing and syntax-controlled text creation. But the inconvenience of this system obscured the potential value of these concepts. For example, if he made a mistake, the user had to reinitialize the file and reconstruct the whole text. To remedy such problems, a text manipulation system was constructed around the basic system. Although some of its features were straightforward adaptations of features in other systems, the complete system was a non-trivial task; it took eight months beyond the four months required for the basic system. This additional work was important because the goal of this project was not only to demonstrate a text creation concept, but also to build a practical tool.

The facilities described below are, in general, called into operation by pushing one of the thirty-two buttons on the program function keyboard. In the descriptions, capitalized names are those that appear over keys on the keyboard overlay (see Appendix A). The descriptions below are divided between text display, text modification, and meta-facilities. The chapter closes with a discussion of two facilities that can be added because Emily text is hierarchically structured.

Text Display Facilities

If an interactive system is the user's primary contact with his information, it must provide a rich variety of mechanisms to view text and locate lower level parts of the structure. There must be some

compensation for the fact that the display size is less than a full page of text. Emily compensates by automatically formatting the display of the text and by providing three means of locating text.

Text displayed by Emily is formatted under control of codes in the syntax tables (see Chapter 5). The user need not--and cannot--insert format control codes in his text. As a result, the system can always properly format text even when the perspective is changed. If low level text is displayed alone, it is indented less and has fewer overflow lines than if it were displayed as part of its parent text. Another result of automatic text formatting is that after a user has learned the format conventions, they help him understand the structure of the text. At the statement level, subordinate text is always indented from its parent. Examples of automatically formatted text are in Figure 2.5 and the appendices.

When viewing a previously created file, the most common operation is holophrast expansion. As described in Chapter 2 and as initially implemented, pointing the light pen at a holophrast caused it to expand one level so that each of its subnodes was a holophrast. While this is theoretically sufficient to observe any view of the file, it proved tedious in practice. Hence, an 'expansion depth' parameter was implemented to control the number of levels that a holophrast is expanded. Normally this parameter is set rather high (21) so that the entire subtree will usually be expanded. But three buttons control the expansion depth for specific purposes. The user can set the depth to one in order to view the immediate subnodes of a holophrast.

A user sometimes wishes to save his place, view some remote text, and return to the original view. To satisfy this user, the Emily system provides display status saving and restoring. The current status of the display can be either stacked or named. In the first option, one button saves the current display on a stack. A second button restores the display status from the top of the stack. Thus the user can travel over the file noting places he wishes to return to and can return simply by pushing a button. The second save-display option is to name the current display. The user pushes the SAVE button and types a name; the RESTORE button displays all names so the user can select the view he wishes next.

A display status is stored as a pointer at the node that generated the display together with some information to partially restore the display to what it looked like. Attempts to preserve the complete display status meet two problems. First, it is necessary to note exactly which nodes are displayed as holophrasts; and second, the text in the saved display might be modified. Detailed holophrast information could be saved in a bit vector with one bit for each node in the order encountered. However, modification of the text could destroy the relation between nodes and bits. The implemented system avoids these problems by saving only a single number. This number is an expansion factor computed by scanning down the tree level by level until the first holophrast is encountered. When the display is restored no subtree is expanded beyond this level. Usually then, the user must manually expand some parts of a restored display. Two other problems were avoided

by careful implementation. If the display generating node or one of its supernodes is moved, the display status still points to the node and when restored will restore the display from wherever the text is located. If the generating node for a display status (or some node above it) is deleted, the display status is also deleted.

A user cannot assign a name to every view he might want to see. On occasion he would like to view those parts of his text containing some specified piece of text. The most general approach to this problem would be to provide a tree search mechanism to find subtrees meeting some description. Though this facility could be added, Emily does not have it. Instead, an 'interactive cross-reference facility' lets the user view all instances of any identifier. A complete description of this facility would require definitions of 'block', 'declaration', and several other terms. Since these definitions are lengthy, they are omitted and the interactive cross-reference facility is only sketched in this paper. The *Emily User's Manual* contains a complete description.

The interactive cross-reference facility is invoked by the user with several buttons. One lets the user select as 'current' an identifier and one of its declarations. Each time a second button is pushed, an instance of the current identifier is displayed. This display is generated by locating the instance and then moving up the structure three levels. (The number of levels can be changed during a console session to display greater or less context.) Buttons are also provided to let the user see the declaration of any identifier or the block

controlling any subblock. In addition to these commands, the stored identifier information controls the choices presented in the menu when the current non-terminal is an identifier. If the non-terminal is a declaration instance, the choice is among all undeclared identifiers in the block or any subblock. Otherwise, the choice is among all identifiers declared in the current block or outer blocks.

Text Modification

As an inevitable by-product of viewing his text, the user will want to modify it. In the earliest version of Emily, the user could delete a holophrast. This changed the holophrast into a non-terminal and saved the deleted structure in a 'dump'. A second button let the user substitute the contents of the dump for the current non-terminal. With this mechanism any text changes can be made. For example, the most common error, especially when copying a routine from paper, is to leave out an operator. The user might input FØØ(3) where he wanted FØØ(3)+X. This change can be made in five steps:

1) contract FØØ(3) to !EXP: FØØ(3)!

2) delete the holophrast to get [<EXP>].
   <EXP> is now the current non-terminal and substitutions for it are displayed in the menu.

3) select for <EXP> the rule
   <EXP> + <EXP>

4) substitute the dump for the first <EXP> to get
   FØØ(3) + <EXP>

5) proceed normally to replace <EXP> with X.

This technique is satisfactory if only one holophrast is involved. But in order to swap two holophrasts, the user would have to temporarily move one of the holophrasts to another non-terminal in the text. To avoid this problem and to provide flexibility in program creation, named fragments were introduced along with the operations of moving and copying subtrees.

A fragment name is an identifier in a special class. Associated with the name is the fragment itself--a subtree of nodes together with an indication of the syntax and non-terminal that generated the subtree. The main text for the file is in a fragment called *MAIN TEXT*. To choose a fragment to work on, the user pushes the EDITFR button; Emily displays the names of existing fragments, the user selects one, and Emily displays the contents of the fragment. The user may then view, add to, or modify the fragment with all the Emily text facilities. The distinction between fragments and display statuses is part of a deliberate separation of text modification and display modification. A fragment is a complete unit of text by itself; it can be deleted, moved, or copied without being displayed. A display status contains a pointer into the text generated by some fragment. To modify the display status would inadvertently modify the fragment. After the user calls a display status to the screen, he can modify the text, but he is modifying the text as part of a fragment and the name of that fragment appears in the message area.

The major text modification functions in Emily are invoked with the MOVE and COPY buttons. After pressing one of these, the user is

asked to light pen the item he wishes to move or copy. He may select a holophrast, an identifier, a constant, or a non-terminal. He may also select a fragment from the menu which displays the names of existing fragments. The item selected is then moved or copied to the fragment with the name *DUMP*. MOVE changes the source item back to a non-terminal, so this button can be used to delete unwanted text. As the next step, both MOVE and COPY ask the user to select a destination for the fragment now in *DUMP*. He may move it to another fragment or back into the text portion of the display.

The item the user initially selects to MOVE or COPY may be any item in the display except a terminal symbol appearing directly in a syntax rule (such as punctuation and keywords). The reason is that such characters are only part of a node, and the whole node may generate many characters. If allowed to MOVE or COPY by pointing at terminals, the user might inadvertently modify more than he wanted. The user can move a node generating a terminal symbol by contracting the node to a holophrast and moving or copying the holophrast.

An example of one use of COPY is shown in Figure 3.1. The fragment TEXT (see Appendix D) was created by first creating the fragment TEMP and then copying it three times. Finally, the <CHAR STR>'s and <NUMBER>'s were filled in appropriately. Though the user must be experienced with Emily before he takes advantage of such possibilities, this example demonstrates text manipulation that cannot easily be accomplished with paper and pencil.

```
Fragment TEMP

   IF TC < '<CHAR STR>' THEN
       CHAINNØ = 1;
   ELSE IF TC ¬> '<CHAR STR>' THEN
       CHAINNØ = <NUMBER> + UNSPEC (TC) - UNSPEC ('<CHAR STR>');
   ELSE <STMT>


Fragment TEXT

   IF TC < 'A' THEN
       CHAINNØ = 1;
   ELSE IF TC ¬> 'I' THEN
       CHAINNØ = 2 + UNSPEC (TC) - UNSPEC ('A');
   ELSE IF TC < 'J' THEN
       CHAINNØ = 1;
   ELSE IF TC ¬> 'R' THEN
       CHAINNØ = 11 + UNSPEC (TC) - UNSPEC ('J');
   ELSE IF TC < 'S' THEN
       CHAINNØ = 1;
   ELSE IF TC ¬> 'Z' THEN
       CHAINNØ = 20 + UNSPEC (TC) - UNSPEC ('S');
   ELSE <STMT>
```

Figure 3.1. Using COPY to Create Text

Both fragments were created from the non-terminal <STMT>. TEMP was created first and copied to TEXT. Then TEMP was copied twice to the final <STMT> in TEXT and the <CHAR STR>'s and <NUMBER>'s were replaced by keyboard entry.

A further facility for text modification is provided by operations on lists. A list is a sequence of elements each generated by the same non-terminal symbol. Non-terminals that can be replaced by lists end with an asterisk; for instance, the non-terminal for a list of statements is <STMT*>. When a list is the current non-terminal, the menu displays options reflecting both the list and the syntax rules for the list-item (e.g. <STMT>). These options provide for continuing or terminating the list while inserting one of the choices for the list-item as an element of the list. For example, to replace |<STMT*>| the menu offers both <ASGN STMT> and <ASGN STMT><STMT*>. In Figure 2.3 each of the two list replacements (<STMT*> → <ASGN STMT> and <ARITH*> → <ARITH>) can be done with a single light pen selection. This reduces the user's interaction time because he can select two rules at once (3-2 and 9-5). Usually the first non-terminal in the inserted choice becomes the new current non-terminal. But there is a special option that inserts the list-item as an element and leaves the list as the current non-terminal. In this case, <STMT*> becomes <STMT><STMT*>. The user thus has the choice of creating the list one element at a time or creating a list of non-terminals and going back to fill them in.

An especially useful feature of lists is that INAFT and DELEL are provided to insert and delete elements. To insert an element, the user pushes INAFT and selects with the light pen the character just before the desired insertion point. If more than one list can have an insertion at that point, the user is asked to resolve the ambiguity by choosing among the non-terminal symbols that generate each of the

possible lists. List element deletion is non-ambiguous, but subnodes must be contracted to a holophrast before they can be deleted. Like a substructure MOVE'd or COPY'd, a deleted list element is moved to *DUMP*.

Meta Facilities

A meta facility is any facility not directly involved in text construction and manipulation. One Emily meta-facility is a provision to extend the utility of the program function keyboard. The 2250 provides a light pen and an alphameric keyboard in addition to the program function keyboard. Since few people have three hands, users must shift from device to device during a console session. As an experiment to avoid this problem and to speed up interaction, program modifications were made so the function keyboard could be used for menu selection and text entry as well as its usual uses. The PFKBMENU, ALPHA SHIFT, and NUM SHIFT buttons set the function keyboard in modes to enter menu selections, letters, and numerals, respectively. This experiment has been unsuccessful so far because the buttons are harder to push than the alphameric keys and because they are arranged so the user must look at them to find the key he wants. Thus this use of the function keyboard distracts the user from the text he is trying to create.

Other Emily meta-facilities enable the user to manipulate files and process entire fragments. These facilities are provided by a monitor routine invoked with the MONITOR button. With the monitor the user can initialize a file and copy one file to another. He may also save the file he is working on and switch to some other file. When he

submits the deck to invoke Emily, the user must include an OS/360 DD card for each file he might want to access. The monitor lets the user choose among these files by displaying whatever names the user has put on the DD cards.

Another monitor facility is a syntax processor that creates syntax tables to control Emily text generation and display. Because the internal coded form of these tables is tedious to generate by hand, the syntactic formalism described in Chapter 5 was designed. Syntax definitions in this formalism are created and modified as Emily texts, then the syntax processor is invoked to convert the definition into syntax tables. Other monitor facilities permit the user to load the new tables and create text in the newly designed syntax.

Fragments can also be printed or punched with the monitor. Printed text is formatted by the same routine that generates the 2250 display, but special caution is taken to avoid excessive indentation. When text would be indented beyond the middle of the page, an extra blank line is printed and the text is brought out to the left margin. Text printed by the system is shown in appendices B, C, and D.

Possible Future Emily Facilities

When text is stored within a computer, programs can be written to process the text and produce desired output. One such program is the Emily syntax processor. Considerable time was saved in the design and implementation of this processor because the text was structured and the existing text access routines could be used. This same fact opens the possibility that routines can be written to modify text.

A text manipulation language can be designed so that an Emily user can write routines to modify his text. To make equality relative, for instance, the user could write a routine to replace all instances of x=y with EQUALS(x,y,EPSILON), where x and y represent arbitrary text structures.

Routines in the text manipulation language can be executed on request as in the example, but they can also be executed when a specified type of node is created or displayed. Several problems in the PL/I syntax could be solved by allowing a subroutine call when specific node types were created. Attributes in declarations could be checked for consistency. After a procedure call had been entered, a routine could check the procedure declaration and create an argument list with non-terminal symbols appropriate for the type of each argument. If a routine could be called when a node were displayed, other problems could be solved. PL/I qualified identifiers could be displayed with only enough qualification to render the reference unambiguous. In outline texts, section numbers could be supplied based on the position of the section in the hierarchy.

Two facilities of value to users could be added to Emily by taking advantage of the graphical display capabilities of the 2250. (This has been avoided in an effort to be compatible with text-only display devices.) First, it is possible to display nodes as blocks with interconnections shown by arrows. This might appear similar to Figure 2.4. Construction of such a mechanism presents interesting problems of formatting a graph structure for display. To be readable, the nodes of the tree would have to be placed so that interconnecting lines were

reasonably short and direct. Once such a mechanism were in Emily, the system could also provide a second facility--'margins' for the user to draw data structure diagrams. With blocks and lines, the Emily user could build sample diagrams of the interconnections of his data structure. While implementing Emily, the author frequently sketched structures in the margin of the coding forms. The display screen would be more flexible, and the resulting diagrams could become part of the documentation of the program. Figure 3.2 shows a possible sketch of part of the Emily data structure. In addition with access to the data structure declarations, Emily could assist in the creation of structure diagrams. For block names, the names of all based structures would be displayed in the menu; for line names, Emily would display the POINTER variables in the based structure. The possibility of such assistance is one more illustration of the value of structured text as used in Emily.

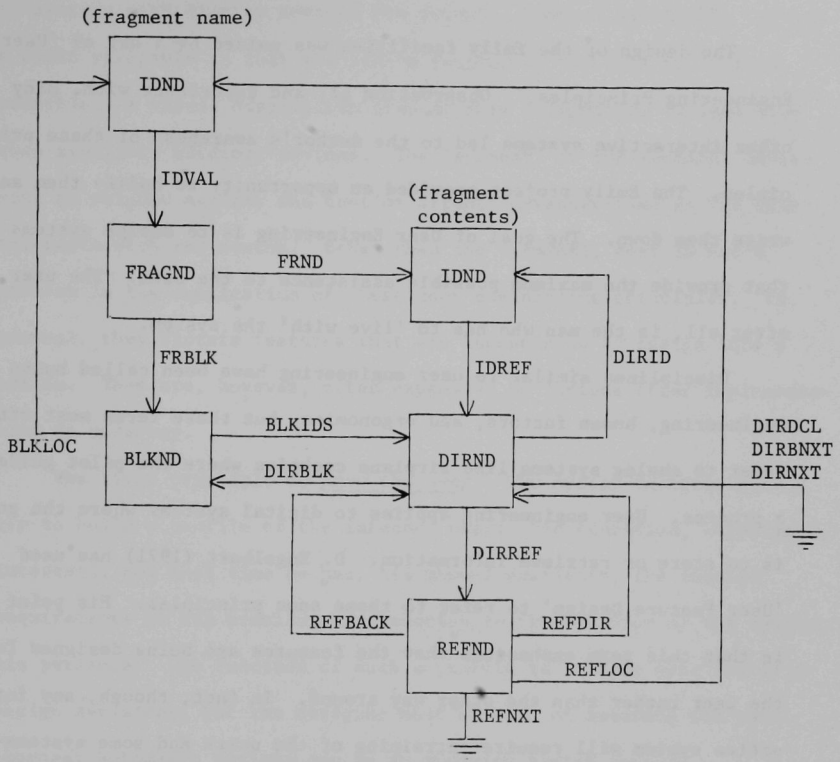


Figure 3.2. Example of a Data Structure Sketch

The proposed structure sketching facility would make possible construction of diagrams like this. The squares represent based structures declared in the text: labeled arrows are pointer variables in those structures. Emily would interpret the user's file to present options of based structure and pointer names. A pointer at an empty list is indicated by ⏚. The structure shown happens to be the Emily data structure created when the contents of a fragment is an identifier.

4. User Engineering Principles

The design of the Emily facilities was guided by a set of 'User Engineering Principles.' Observation of, and experience with, many other interactive systems led to the author's awareness of these principles. The Emily project provided an opportunity to codify them and write them down. The goal of User Engineering is to design systems that provide the maximum possible assistance to the user. The user, after all, is the man who has to 'live with' the system.

Disciplines similar to user engineering have been called human engineering, human factors, and ergonomics, but these terms most often refer to analog systems like airplane cockpits where the pilot guides a process. User engineering applies to digital systems where the goal is to store or retrieve information. D. Engelbart (1971) has used 'User Feature Design' to refer to these same principles. His point is that this term emphasizes that the features are being designed for the user rather than the other way around. In fact, though, any interactive system will require retraining of the users and some systems--like Emily--may require the user to alter thinking habits of many years standing. But let there be no mistake, the author is deeply committed to a policy of modifying the system to fit the user. Other sets of user engineering principles have been reported by L. B. Smith (1969) and J. G. Mitchell (1970). Their suggestions are compatible with those below, but less comprehensive. The reader should also read R. B. Miller's paper (1968) in which he attempts to estimate a maximum permissible response time in seventeen interactive contexts.

One restriction on a few of the principles below is that they apply to systems with display devices for output. This is essential, because a basic principle is that the system respond to the user as fast as possible. A visual display can present more information in less time than available hardcopy devices. The 'economy' of the terminal device must be weighed against the cost of attention-wander-time as the user interacts with the system. Other than the terminal, cost is not a problem in the application of these user engineering principles. In general, they dictate features that are inexpensive to design into a system. They are, however, often expensive to include after implementation is underway.

The first principle is Know the user. The system designer should try to build a profile of the intended user: his education, experience, interests, how much time he has, his manual dexterity, the special requirements of his problem, his reaction to the behavior of the system, his patience. One function of such a profile is to help make specific design decisions, but the designer must be wary of assuming too much. Improper automatic actions can be an annoying system feature.

A more important function of the first principle is to remind the designer that the user is a human. He is someone to whom the designer should be considerate and for whom the designer should expend effort to provide conveniences. Furthermore, the designer must remember that human users share two common traits: they forget and they make mistakes. With any interactive system problems will arise--whether the user is a high school girl entering orders or a company president asking for a sales breakdown. The user will forget how to do what he wants,

what his files contain, and even--if interrupted--what he wanted to do. Good system design must consider such foibles and try to limit their consequences. The Emily design tried to limit these consequences by explicitly including a fallible memory and a capacity for errors in the intended user profile. Other characteristics assumed are:

curious to learn to use a new tool,

skilled at breaking a problem into sub-problems,

familiar with the concept of syntax and the general features of the syntax for the language he is using,

manually dextrous enough to use the light pen,

not necessarily good at typing.

Specific user engineering principles to help meet the first principle can be categorized into

minimize memorization,

optimize operations,

engineer for errors.

The principles are outlined in Figure 4.1.

Minimize Memorization

Because the user forgets, the computer memory must augment his memory. One important way this can be accomplished is by observing the principle selection not entry. Rather than type a character string or operation name, the user should select the appropriate item from a list displayed by the computer. In a sense, the entire Emily system is based on this principle. The user selects syntax rules from the

User Engineering Principles

First principle: Know the user

Minimize Memorization

- Selection not entry
- Names not numbers
- Predictable behavior
- Access to system information

Optimize Operations

- Rapid execution of common operations
- Display inertia
- Muscle memory
- Reorganize command parameters

Engineer for Errors

- Good error messages
- Engineer out the common errors
- Reversible actions
- Redundancy
- Data structure integrity

Figure 4.1. Table of User Engineering Principles

menu and never types text. Even when an identifier is to be entered, Emily displays previously entered identifiers: though the user must type in new identifiers. Because the system is presenting choices the user need not remember the exact syntax of statements in the language, nor the spelling of identifiers he has declared. Moreover, each selection--a single action by the user--adds many characters to the text. Thus if the system can keep up with the user, he can build his text more quickly than by keyboard entry.

The principle of 'selection not entry' is central to computer graphics and by itself constitutes a revolution in work methods. The fact is that a graphic display--attached to a high bandwidth channel--can display many characters in the time it would take a user to type very few. If the choices displayed cover the users' needs, he can enter information more quickly by selection. The author first saw the principle in (Smith, 1969) and (George, 1968) but has since observed it in many systems. Noteworthy examples are an order entry system for a mail-order company (Gladwin, 1970) and a hospital patient-note system (Ridsdale, 1970).

In these latter two systems, selection is not by light pen but by code entry through an alphameric keyboard. Experience with Emily suggests that keyboard code entry is better than light pen selection because of two user frustrations. First, the menu does not provide a target for the light pen while the display is changing: and second, the delay can vary depending on system load. With keyboard codes, the user can go at full speed in making selections he is familiar with, but when he gets to unfamiliar situations he can slow down and wait

for the display. Thus, his behavior can travel the spectrum from typing speed to machine paced selection.

The second principle to avoid memorization is names not numbers. When the user is to select from a set of items he should be able to select among them by name. In too many systems, choices are made by entering a number or code which the system uses to index into a set of values. Users can and do memorize the codes for their frequent choices, though this is one more piece of information to obscure the problem at hand. But when an uncommon choice is needed, a code book must be referenced. Symbol tables are understood well enough that there is no excuse for not designing them into systems so as to replace code numbers with names. In Emily, there are names for files, fragments, display statuses, syntaxes, and non-terminals. Conceivably, the user could even supply a name to be displayed in each holophrast. In practice, though, so many holophrasts are displayed that the user would never be done making up names. For this reason, the holophrast contains the non-terminal and the first few characters for the text--a system generated 'name' with a close relation to the information represented by that name. Because it is also possible to forget the meaning of a name, a system should also provide a dictionary. System names should be predefined and the user should be allowed to annotate any other names he creates. The lack of a dictionary in Emily has sometimes been a nuisance while trying to remember what different text fragments contain.

The next principle, predictable behavior, is not easy to describe. The importance of such behavior is that the user can gain an 'impression'

of the system and understand its behavior in terms of that impression. Thus by remembering a few characteristics and a few exceptions, the user can work out for himself the details of any individual operation. In other words, the system ought to have a 'Gestalt' or 'personality' around which the user can organize his perception of the system. In Emily all operations on text appear to make it expand and contract. Text creation expands a non-terminal to a string and the viewing operations expand and contract between strings and holophrasts. This commonality lends the unity of predictable behavior to Emily.

Predictable behavior is also enhanced by a modular system designed in accordance with the hierarchical approach. If the same subroutine is always used for some common interaction, the user can become accustomed to the idiosyncracies of that interaction. For instance, in Emily there is one subroutine for entering names and other text strings so that all keyboard interactions follow the same conventions.

The last memory minimization principle is access to system information. Any system is controlled by various parameters and keeps various statistics. The user should be given access to these and should be able to modify from the console any parameter that he can modify in any other way. With access to the system information, the user need not remember what he said and is not kept in the dark about what is going on. Emily provides means of setting several parameters, but fails to have any mechanism for displaying their values. This oversight is due to a failure to remember that the user might not have written the system. Another such oversight is a failure to provide

error messages for many trivial user errors. Even worse, the 'MULTIPLE DECLARATION' error message originally failed to say which identifier was so declared. This has been corrected, but should have been avoided by attention to the 'Access to system information' principle of user engineering.

Optimize Operations

The previous section stressed the design--the logical facilities--of the set of commands available to the user. 'Optimize operations' stresses the physical appearance of the system--the modes and speeds of interaction and the sequence of user actions needed to invoke specific facilities. The guiding principle is that the system should be an unobtrusive as possible, a tool that is wielded almost without conscious effort. The user should be encouraged to think not in terms of the light pen and keyboard, but in terms of how he wants to change the displayed information.

The first step in operation optimization is to design for *rapid execution of common operations*. Because Emily text is frequently modified in terms of its syntactic organization, a data structure to represent text was chosen so as to optimize such modification. The text display is regenerated frequently, so considerable effort was expended to optimize that routine. More effort is required, though; it is still slow largely because a subroutine is called to output each symbol. Less frequent operations like file switching do not justify special optimization. Lengthy operations, however, should display occasional messages

to indicate that no difficulty has occurred. For instance, while printing a file Emily displays the line number of each tenth line as it is printed.

As the system reacts to a user's request, it should observe the principle of display inertia. This means the display should change as little as necessary to carry out the request. The Emily DELETE operation replaces a holophrast (and the text it represents) with a non-terminal symbol. The size and layout of the display do not change drastically. Text cannot be deleted without first being contracted to a holophrast, thus deletion--a drastic and possibly confusing operation--does not add the disorientation of a radically changed display. The Emily display also retains inertia in that the top line changes only on explicit command. Some linear text systems always change the display so the line being operated on is in the middle of the display. Because the top line keeps changing, the user is sometimes not sure where he is in the text. The Emily automatic indentation provides additional assistance to the user. As text is created in the middle of the display, the bottom line moves down the display. Since this line is often not indented as far as the preceding line, its movement makes a readily perceptible change in the display.

One means of reducing the user's interaction effort is to design the system so the user can operate it on 'muscle memory.' Very repetitive operations like driving a car or typing are delegated by the conscious mind to the lower part of the brain (the medulla oblongata). This part of the brain controls the body muscles and can be trained to perform operations without continual control from the conscious mind.

One implication of muscle memory is that the meaning of specific interactions should have a simple relation to the state of the system. A button should not have more than a few state dependent meanings and one button should be reserved to always return the system to some basic control state. With such a button. the muscle memory can be trained to escape from any strange or unwanted state so as to transfer to a desired state. In Emily the buttons of the program function keyboard obey these principles. The NORMAL button always returns the entire system to a basic state waiting for commands. Other buttons have very limited meanings and it is usually possible to abort one command and invoke another simply by pushing the other button (without pushing NORMAL first).

A second implication of muscle memory for system design is that the system must be prepared to accept commands in bursts exceeding ten per second. (Typing 100 words per minute is 10 characters per second. A typing burst can be faster.) It is not essential that the system react to commands at this rate, because interactive computer use is characterized by command bursts followed by pauses for new inspiration. But if command bursts are not accepted at a high rate, the muscle memory portion of the brain cannot be given full responsibility for operations. The conscious brain has to scan the system indicators waiting for GO. Command bursts from muscle memory account for the unsuitability of the light pen for rule selection as discussed under 'selection not entry'.

Another failure of Emily to accommodate muscle memory is the mobility of identifiers in the menu. As new identifiers are created

they are inserted in the menu in alphabetic order. The user cannot memorize the location of an item, but must read the menu each time to find what he wants, if it is there at all. At least two solutions are possible and perhaps both are needed. The first solution would be to display the letters of the alphabet for initial selection and then to display the identifiers available in that section of the alphabet. This would only be done if there were more than two menu-fulls of identifiers so that in many cases a user would only have to make one interaction to select an identifier. As a second solution, special menus would be available whose contents were entirely under the user's control. The user would be able to store identifiers and fragments in these special menus and recall them with button pushes. Since he has control of the location in the menu, he can find a required item very quickly.

In addition to optimizing the interaction time, the system designer must be prepared to reorganize command parameters. Observation of users in action will show that some commands are not as convenient as their frequency warrants while other commands are seldom used. Inconvenient commands can be simplified while infrequent commands can be relegated to sub-commands. Such reorganization is simplified if the original system design has been adequately modularized. High level command routines can be rewritten without rewriting low level routines and the latter can be used without fear that they depend on the higher level.

A good example of command reorganization in Emily has been the evolution of the view expansion commands. In the earliest version, pointing the light pen at a holophrast expanded it one level, so that each of the subnodes of the holophrast became a new holophrast. With this mechanism, many interactions were required to view the entire structure represented by a holophrast. Very soon the system-designer/user added a system parameter called 'expansion depth'. This parameter dictated how many levels of a holophrast were to be expanded. To set the expansion depth, the user pushed a button (on the program function keyboard) and typed in a number (on the alphameric keyboard). It soon became obvious that users almost always set the expansion depth to either one or all. Consequently, two buttons were defined, so that the user could choose either option quickly. Later, the button for typing in the expansion depth was removed and that function placed under a general 'set parameters' command. Further experience seems to indicate that only the 'expand one level' button is required. It would take effect only during the next holophrast expansion. At all other times, holophrasts would always be expanded as far as possible.

Engineer for Errors

Modern computers can perform billions of operations without errors. Knowing this, system designers tend to forget that neither users nor system implementers achieve perfection. The system design must protect the user from both the system and himself. After he has learned to use a system, a serious user seldom commits a deliberate error. Usually he

is forgetful, or pushes the wrong button without looking, or tries to do something entirely reasonable that never occurred to the system designer. The learner, on the other hand. has a powerful, and reasonable, curiosity to find out what happens when he does something wrong. A system must protect itself from all such errors and, as far as possible, protect the user from any serious consequences. The system should be engineered to make catastrophic errors difficult and to permit recovery from as many errors as possible.

The first principle in error engineering is to provide good error messages. These serve as an invaluable training aid to the learner and as a gentle reminder to the expert. With a graphic display it is possible to present error messages rapidly without wasting the user's time Error messages should be specific, indicating the type of error and the exact location of the error in the text. Emily does not have good messages for user errors. Currently, the system blows the whistle on the 2250 and waits for the next command from the user. Each error is internally identified by a unique number, and it will not be difficult to change the system to display the appropriate message for each number.

It is not enough to simply tell the user of his errors. The system designer must also be told so he can apply the principle engineer out the common errors. If an error occurs frequently, it is not the fault of the user, it is a problem in the system design. Perhaps the keyboard layout is poor or commands require too much information. Perhaps

consideration must be given to the organization of basic operations into higher level commands.

Emily provides several means of feedback from the user to the system designer. (Though for the most part, they have been one and the same.) A log is kept of all user interactions, user errors, and system errors. There is a command to let the user type a message to be put in the log and this message is followed by a row of asterisks. When the user is frustrated he can push a 'sympathy' button. In response, Emily displays at random one of ten sympathetic messages. More importantly, frustration is noted in the log and the system designer can examine the user's preceding actions to find out where his understanding differed from the system implementation.

'Engineering errors out' does not mean to make them impossible. Rather they should be made sufficiently more difficult that the user must pause and think before he errs. In Emily, time consuming operations like file manipulation always ask the user for additional operands. If he does not want the time consuming operation he can do something else. To delete text, the user must think and contract it to a holophrast. This means that large structures cannot be cavalierly deleted.

A single erroneous deletion can inadvertently remove a very large substructure from the file. To protect the user the system must provide reversible actions. There ought to be one or more well understood means for undoing the effects of any system operation. In Emily, a deleted structure is moved to *DUMP*. If the user has made a mistake, he can reach into this 'trash can' and retrieve the last structure he

has deleted. (Deletion does destroy the old contents of *DUMP*.) A more general reversible action mechanism would be a single button that always restored the state existing before the last user interaction. Emily has no such button, but the QED system (Thompson, 1968) supplies a file containing all commands issued during the console session. The user can modify this file of commands and then use it as a source of commands to modify the original text file again.

Besides helping the user escape his own mistakes, error engineering must protect the user from bugs in the system and its supporting software. Modular design is important to such protection because it minimizes the dependencies among system routines. The implementer should be able to modify and improve a routine with confidence that his changes will affect only the operation of that routine. Even if the changes introduce bugs, the user will be protected if the designer has observed the principles of redundancy and data structure integrity.

Redundancy simply means that the system provides more than one means to any given end. A powerful operation can be backed up by combinations of simpler operations. Then if the powerful operator fails, the user can still continue with his work. Such redundancy is most helpful while debugging a system, but very few systems are completely debugged and any aids to the debugger can help the user. As an adjunct of redundancy, the system must detect errors and let the user act on them, rather than simply dumping memory and terminating the run. In Emily, the PL/I ON-condition mechanism very satisfactorily catches errors. They are passed to a subroutine in Emily that tells the user that a catastrophe has occurred

and names the routine where the error was detected. Control then returns to the normal state of waiting for a command from the user, who has the option to continue or call for a dump.

A system should provide sufficient data structure integrity that regardless of system or hardware trouble some version of the user information will always be available. This principle is especially applicable to Emily where most of the information is encoded by pointers. A small error in one pointer can lose a large chunk of the file. Some effort has been spent ensuring that errors in Emily will not damage the part of the data structure kept in core during execution. But if an error abruptly terminates Emily execution (such errors are generally in the system outside Emily) the file on the disk may be in a confused state. Currently, the only protection is to copy the file before changing it, but there are file safety systems that do not rely on the user to protect himself.

In a safer system, modified text records would be written to a temporary file. The rest of the file would be copied to the temporary file when the user switched to another file. Then the temporary file and the original file would be renamed so as to swap roles. With this system, there would always be one correct copy of the file, even if it was not completely up-to-date. For added protection and to optimize access, an independent program can be written to reorganize the data structure in an Emily file. This reorganization would attempt to put a node and its subnodes all in the same record. An algorithm for

reordering a data structure to localize substructures has been reported in 'Compact List Representation: Definition, Garbage Collection, System Implementation' (Hansen, 1969).

Protection and assistance for the user are keywords in user engineering. The principles outlined in this chapter are not as important as the general approach of tailoring the system to the user. Only by such an approach can Computer Science divest the computer of its image as a cold, intractable, and demanding machine. Only by such an approach can the computer be made sufficiently useful and attractive to take its place as a valuable tool for the creative worker.

5. Syntactic Formalism

The potential of the current Emily implementation is described and circumscribed by the syntactic formalism in which user languages are described. This chapter outlines that formalism in sufficient detail for the reader to understand its limitations. Definition of several terms will facilitate further discussion:

user language - the language being described in the formalism. An Emily user will create text in this language.

Emily formalism (or just formalism) - a language for describing user languages.

syntax definition - a statement written in the formalism to describe a specific user language.

syntax designer - the expert who creates a syntax definition.

Abstract and Concrete Syntax

The formal description of the user language must fulfill three functions within the Emily system. First, it specifies the internal structure that will represent a tree. For this purpose, the syntax must specify a sequence of non-terminals for the right side of every rule. Second, the syntax must specify how each rule is to be displayed. Not only must this include terminal symbols, but it must include control of indentation and other formatting. Finally, a simple string must be generated for each rule to represent that rule as an option in the menu. This function is performed by assuming that each subnode of the rule is a non-terminal symbol.

To satisfy these three functions, the syntax is coded into a pair of tables. One table--the abstract syntax--controls the hierarchical structure of generated text. It specifies which syntax rules can replace a given non-terminal symbol and the sequence of non-terminal symbols on the right side of each syntax rule. Another table--the concrete syntax--tells how to display each rule: it includes punctuation, reserved words, and formatting information like indentation and line termination. The division into abstract and concrete syntax, first suggested by J. McCarthy (1963), means that there can be more than one concrete syntax for each syntax and thus that a text can be displayed in more than one format. This has such applications as emphasizing certain text components, changing from partial to full parenthesization. and even partially converting from one language to another. A language definition in the formalism is converted into the abstract and concrete syntax tables by the syntax processor.

The syntactic formalism permits several extensions to the normal BNF descriptive formalism, as was used in Chapter 2. In this chapter, the Emily formalism is described by means of the BNF formalism. But because BNF does not provide a convenient method of specifying the spacing and indentation of the text, the Emily formalism is described by means of itself in Appendix B.

Structure of the Formalism

Like BNF, the Emily formalism is a sequence of rules specifying substitutions for non-terminal symbols. The essential components of a syntax definition can be described (in BNF) by:

```
⟨syndef⟩ ::= ⟨pr⟩*
⟨pr⟩     ::= < ⟨nt⟩ > ⟨rhs⟩*
⟨rhs⟩    ::= : ⟨item⟩*
⟨item⟩   ::= < ⟨nt⟩ > | '⟨string⟩'
```

Here, the symbols have these meanings:

⟨syndef⟩ - syntax definition;

⟨pr⟩ - (production) a non-terminal and a set of possible replacements·

⟨rhs⟩ - (right-hand-side) a string of symbols representing one possible replacement;

⟨item⟩ - one of the symbols in the right-hand-side;

⟨nt⟩ - (non-terminal) a non-terminal in the description of the user language;

⟨string⟩ - a terminal symbol in the user language.

The superscript asterisks indicate that those non-terminal symbols may appear one or more times. Note that the non-terminal brackets, '<' and '>', are explicit. When the syntax designer enters a non-terminal name, he does not enter these brackets. (Readers familiar with BNF may notice that this syntax is ambiguous. The system is unaware of ambiguity though because text is generated and not parsed. For the reader, the ambiguity is resolved by indentation.)

One possible syntax written in the Emily formalism is:

```
<AEXP> : <ATERM> '+' <AEXP>
       : <ATERM> '-' <AEXP>
<ATERM> : <ARITH>
        : '(' <AEXP> ')'
```

The reader might convince himself that the ⟨item⟩'s in this syntax are six instances of <⟨nt⟩> and four instances of '⟨string⟩'.

A more complete description of the Emily formalism in shown in Figure 5.1. Careful comparison of the figure with the simple description above will show that the latter is indeed a subset of the former. Note in the figure that the rule for ⟨syndef⟩ and the fourth choice for ⟨pr⟩ allow the syntax designer to group the syntax in sections each with its own ⟨title⟩. This means that rather than just a sequence of rules, the syntax definition may itself be a hierarchical structure. This is a convenience for the designer and readers of the syntax, but does not affect the expressive power of the formalism.

The abstract syntax tables are generated solely from the structure generated by ⟨pr⟩'s, ⟨rhs⟩'s, ⟨star⟩'s, and those ⟨item⟩'s that contain ⟨nt⟩. These components will be discussed in the next two sections. All other ⟨item⟩'s are included solely for generation of the concrete syntax, as discussed in the following two sections. A final section evaluates this formalism with respect to its impact on the user and the system designer.

```
⟨syndef⟩ ::= ⟨title⟩ < ⟨nt⟩ > IS OUTER N-T
                ⟨pr⟩* ⟨printspec⟩*?

⟨pr⟩ ::= < ⟨nt⟩ > ⟨rhs⟩*
       | < ⟨nt⟩ > IS A CONSTANT
       | < ⟨nt⟩ > IS AN ID
       | ⟨title⟩ ⟨pr⟩* ⟨printspec⟩*?

⟨printspec⟩ ::= < ⟨nt⟩ *> ⟨rhs⟩

⟨rhs⟩ ::= : ⟨item⟩* | : (⟨label⟩) ⟨item⟩*
        | : BLOCK ⟨item⟩*
        | : (⟨label⟩) BLOCK ⟨item⟩*

⟨item⟩ ::= < ⟨nt⟩ > | < ⟨nt⟩ / ⟨star⟩ >
         | < ⟨nt⟩/DECL > | < ⟨nt⟩/DECL/⟨star⟩ >
         | '⟨string⟩' | INDENT ⟨indent⟩ | NL?
         | IFT (⟨item⟩*) | NOT
         | SND ⟨sndno⟩ IN (⟨label⟩ - ⟨label⟩)
         | DAD = (⟨label⟩) | SNDMT ⟨sndno⟩
         | XIT | ... (18 more options)

⟨star⟩ ::= * | *? | ? | */⟨item⟩*
         | *?/⟨item⟩* | */'⟨string⟩'
         | *?/'⟨string⟩'
         | */(⟨label⟩) | *?/(⟨label⟩)
```

Figure 5.1. BNF Description of Emily Syntactic Formalism

A superscript '*' on a non-terminal symbol means that symbol may be repeated one or more times; '*?' indicates zero or more repetitions. The following represent non-terminals that must be replaced by terminal strings typed on the keyboard: ⟨title⟩, ⟨nt⟩, ⟨label⟩, ⟨string⟩, ⟨indent⟩, and ⟨sndno⟩.

Identifiers

The second and third options for ⟨pr⟩ in Figure 5.1 declare non-terminal symbols to represent classes of terminal symbols. For example

```
<NUMBER> IS A CONSTANT
<ARITH> IS AN ID
```

declare that the non-terminals <NUMBER> and <ARITH> must be replaced by terminal symbols entered from the keyboard. Every non-terminal specified to be a CONSTANT or an ID generates a different class of symbol. For all practical purposes, all CONSTANT's are treated alike. Every CONSTANT entered is stored separately. Identifiers, though, are stored uniquely in each class. That is, there is one copy of the identifier and every instance of it is stored as a pointer to that single copy. Identifiers can be accessed with the interactive cross reference facility.

The identifier facility has been implemented so identifiers obey the declaration block scope rules of PL/I and Algol. This feature requires considerable code in the system and the options BLOCK and DECL in the formalism. Of the options for ⟨item⟩, DECL appears in two of the four choices containing ⟨nt⟩. The DECL option is used only when the ⟨nt⟩ is elsewhere specified to be an identifier. Instances of identifiers generated by <⟨nt⟩/DECL> or <⟨nt⟩/DECL/⟨star⟩> are recognized as declaration instances. All other identifier instances are recognized as identifier references. For example, in a syntax for PL/I there might appear

```
<DECL>   : DCL <ARITH/DECL> FIXED;
<ASSIGN> : <ARITH> = <ARITHX>;
```

Then if the generated text contains

```
DCL A FIXED; A = <ARITHX>;
```

the system recognizes that the first instance is a declaration and the second is a reference to the declared identifier.

Block scope is controlled by the BLOCK options for ⟨rhs⟩. Any node generated from a rule with this option delimits a block. All declarations in that node and its subnodes control only references within the same set of nodes. In PL/I, a procedure name is in the scope outside the procedure but any parameter names are within that scope. For this reason, the syntax for procedures looks like this:

```
<PROC> : <ENTRYNM> : PROC <PROCBODY> END;
<PROCBODY> : BLOCK <PROCOPT/*/' '>; <STMT/*/NL?>
<PROCOPT> : (<PARM/*/', '>)
```

All identifiers declared within a <PROCBODY> are accessible only within the <PROCOPT>'s and the <STMT>'s.

Lists

Because lists of various kinds are found in most languages, it is convenient to have a special representation for lists in any syntactic formalism. In Figure 5.1, a superscript asterisk indicates that the given non-terminal may be repeated any number of times. In the Emily formalism, lists are represented by items in one of these two forms:

<⟨nt⟩/⟨star⟩> <⟨nt⟩/DECL/⟨star⟩>

where ⟨star⟩ has nine options, each generating a slightly different kind of list. The most elementary is the plain asterisk as in

```
<STMT/*>
```

which represents a sequence of one or more statements. Asterisk and question mark indicates a sequence of zero or more elements. Question mark alone describes a non-terminal that may appear zero or one times. In practice, this means the user is given the choice EMPTY along with all other choices for the indicated non-terminal. If he selects EMPTY, the non-terminal simply disappears from the text.

Some options for ⟨star⟩ contain a slash. These options specify the separator symbols that will appear between successive non-terminals in the list. For instance, <PARM/*/', '> specifies a list of one or more parameters separated by commas. The separator may be any sequence of items; sometimes it is convenient to specify format codes in addition to character strings. Because non-terminals can be included as separators, a possible syntax for arithmetic expressions could be

```
<ARX> : <ARX/*/<AOP>>
      : <VAR>
      : (<ARX>)
<AOP> : +
      : *
```

However, it is difficult to automatically parenthesize such expressions. A list separator is often just a string of characters, so this possibility is provided as a shortcut. The string in <PARM/*/', '> can be

generated with either */⟨item⟩* or the shortcut */'⟨string⟩'.

From a list item, the syntax processor generates a unique non-terminal with a single replacement rule. Ordinarily this rule contains the list element non-terminal, the separator sequence, and the unique list non-terminal. To permit the syntax designer to specify the entire contents of this rule, the ⟨printspec⟩ mechanism is provided. The designer chooses a (⟨label⟩) option for ⟨star⟩ and elsewhere generates a ⟨printspec⟩ with the list element non-terminal on the left and the same label on the right (as part of the ⟨rhs⟩). The right side of this ⟨printspec⟩ is taken as the syntax rule for the list non-terminal. This mechanism was used in the PL/I syntax to generate a list of <ENTRYNM>'s each preceded by a colon.

Display Format

All ⟨item⟩'s not discussed above are provided only to describe the display format of the generated text. Text for a node is generated from left to right. When a ⟨string⟩ is encountered, it is displayed as the next piece of text. When an <⟨nt⟩> is encountered, a check is made to see whether the user has created the corresponding subnode yet. If not, the non-terminal symbol is displayed, otherwise, the display generator calls itself (recursively) to generate text for the subnode. On return, generation of text continues with the next item in sequence for the current node.

Items other than non-terminals and strings can be viewed as 'operators' that are 'executed.' Thus the syntax is not solely a descriptive

mechanism, but specifies a sequence of actions that occur in a given order. This view of syntax is similar to that used in the Meta systems (Schorre, 1964) where the sequence of actions is used to specify a translation of a string recognized by the syntax.

The most fundamental (and first implemented) Emily display format operators are NL? and INDENT. NL? specifies that the current display should be terminated and further text should be displayed on the next line. If the text is already at the beginning of a line, though, NL? has no effect. (Thus the question mark indicates uncertainty rather than a predicate.) Any indentation of the new line is controlled by prior execution of the INDENT operator. The indentation specified with INDENT is relative to the indentation when the current node was entered. Thus the indentation can be increased for part of the display of a node and decreased thereafter. As an illustration, a DO-group might have the syntax:

```
<STMT> : 'DO' INDENT+8 <DO OPT>
           INDENT+4 NL? <STMT/*/NL?>
           INDENT+0 NL? 'END;'
```

Here if the DO-OPTion extends to more than one line it will be indented eight spaces while the list of statements will be indented four spaces. Finally, the 'END;' will be at the same indentation as the 'DO'.

Conditional Display

Many display formats cannot be described with the above mechanisms. For these Emily provides conditional format control operations. The syntax designer can test the contents of a given subnode to determine how to display the text. Conditional display is based on a toggle switch (a different switch for each recursive level of the display generator). Several display operators set the toggle true or false depending on text being displayed. Then the operation

IFT (⟨item⟩*)

tests the toggle and executes the enclosed sequence of ⟨item⟩'s only if the toggle is true. Multiple conditions can be tested by nesting IFT(...) operations.

The choices for ⟨item⟩ in Figure 5.1 show four of the toggle setting operators. NOT simply changes the setting of the toggle from true to false and vice versa. In the option

SND ⟨sndno⟩ IN (⟨label⟩ - ⟨label⟩)

the two ⟨label⟩'s must appear on rules for the ⟨sndno⟩'th non-terminal symbol in the current node. The toggle is set true if that non-terminal has been replaced by a node generated according to one of the rules between the two labels. To illustrate, note that an addition must be parenthesized if it is an operand of a multiplication. This is described in the formalism as follows:

```
<ARITHX> : <ARITH>
         : (ADD) <ARITHX> '+' <ARITHX>
         : (SUB) <ARITHX> '-' SND 2 IN (ADD-ADD) IFT('(') <ARITHX> IFT(')')
         : (MUL) SND1 IN (ADD-SUB) IFT('(') <ARITHX> IFT(')')
           '*' SND2 IN (ADD-SUB) IFT('(') <ARITHX> IFT(')')
```

Either subnode of a (MUL) will be parenthesized if it is an addition or a subtraction.

The test SNDMT ⟨sndno⟩ sets the toggle true if the ⟨sndno⟩'th subnode has been replaced by EMPTY. (This is possible only if the subnode is described with <⟨nt⟩/⟨star⟩> and the star option contains a question mark.) This test for EMPTY is used to control the display of list separators. When the syntax designer codes--for example, <ARG/*/', '> the syntax processor generates a production for the non-terminal <ARG*>. This production has a form that could be described by

```
<ARG> SNDMT2 IFT(XIT)', '<ARG*>  .
```

The XIT operator used here simply discontinues execution of operators for this node. Control returns to the next operator for the node containing the current node.

A final conditional display test is the operator

```
DAD = (⟨label⟩)  .
```

This operator sets the toggle true only if the father of the current node was generated by the rule with the given label on its right hand side. This permits limited context testing to control the display of text.

Evaluation of Syntactic Formalism

Initially, Emily was to be implemented for PL/I only and there were no plans to design a syntactic formalism. It soon became apparent, though, that the syntax would have to be adjusted and modified. Since the system was table driven anyway, it seemed possible to build a compiler to translate from a syntax definition to internal tables. With each change in system facilities, new features were added to the formalism. Lists, conditional display operations, and identifier scopes each required modification to both the formalism and the compiler. As will be discussed in the next paragraphs, the resulting formalism meets the goals of preventing errors and handling ambiguous syntaxes, but there are significant limitations.

The original vision of the Emily system pictured a system that would not permit the user to make any syntax errors. Not only would he always have punctuation correct, but the system would also segregate identifiers into classes and permit only valid identifiers in expressions. In practice, such benevolent protectionism is difficult to achieve. A particular problem is that PL/I allows factored declarations. Attempts to protect against incorrect attributes at any level of factoring lead to a bewildering variety of non-terminal symbols. This is a manifestation of a more general problem: as the specificity of the syntax increases, so do the number of choices and thus the number of interactions required by the user to create a given text. Only with a more extensive computational capability could a syntactic formalism

prevent all possible errors. Because of the expense--in both core and time--such checking is perhaps best left to compilers. The interactive system should attempt only to reduce errors to the extent that all can be detected with one compilation.

Most systems based on BNF must make one or another restriction on the grammar so the system can generate a recognizer to parse strings in the language. With Emily, strings are always parsed and always meet the grammar. Thus, as was expected when the project began, Emily can handle any syntax, no matter how ambiguous. But texts in the grammar must be read by the user and understood. For this purpose, the display of the text, at least, has to be unambiguous. One problem that arose early was that the text could appear unambiguous to both the system and the reader and yet still be read wrong by the PL/I compiler. Consider

IF bool-exp THEN stmt-1 ELSE stmt-2
and IF bool-exp THEN stmt-3.

If stmt-1 is replaced with the second form of IF statement, the PL/I rules specify that the ELSE goes with the second IF. But Emily would be aware of the structure and would display the inner IF indented from both the outer IF and its ELSE. The user would tend to read the statement as he intended it instead of as the PL/I compiler will read it. This problem was solved with the conditional facilities of the syntax display mechanism. In the above case, the ambiguity is resolved by automatically generating an ELSE after the inner IF statement.

Experience has shown that the Emily formalism should have had more provision to test the typographical environment. Currently, the environment can only be tested to determine if the next character will be placed beyond some specified column. It would sometimes be convenient to test how many characters are left in the current line or how many lines are left in the display or the page. These and other environment tests can be added to the display format mechanisms with little modification to the current system.

The Emily display mechanism can be contrasted with the mechanism used by Koch and Schwarzenberger (1969) to format the printing of syntax rules for PL/I. In their formalism, the display of a node can be controlled by two levels of context. That is, the choice of format can be based on both father and grandfather. Unlike Emily, subnodes cannot be tested. The Emily mechanism was chosen because it was simple to implement. It requires only one display format for each rule while the Vienna mechanism requires several with choice being made on the basis of context.

A major limitation of the Emily syntactic formalism is that one syntax specification is used for three purposes: to describe the structure of the generated text, to control the display of that text, and to specify the choices to be presented in the menu. To a degree, the display can be modified by replacing the concrete syntax, but the menu and text structure are inextricably linked. If the syntax designer desires to change the order of menu choices, he cannot use the new

syntax for any existing text. Moreover, if he provides 'shortcut' productions that will be the same as some tree of other productions, the two trees will be different internally, and the user cannot dissect the shortcut generated tree into its components. A third problem is that some sets of choices are valid for several different non-terminals. For example seven attributes are repeated for each of <ARITH ATTR>, <BIT ATTR>, <CHAR ATTR>, <PTR ATTR>, and <DATA ATTR>. This repetition requires a substantial amount of table space.

To solve these problems, the distinction between abstract and concrete syntaxes must be extended to three syntax specifications. The basic syntax would specify a set of replacement strings where each string included only non-terminal symbols. Strings would be labeled so they could be referenced from the other two syntaxes and so replacements with only terminal symbols could be distinguished from each other. A second syntax would specify a display format for any tree constructed according to the basic syntax. These first two syntaxes are roughly equivalent to the current abstract and concrete syntaxes. The third syntax would specify the choices in the menu. For each choice, a menu representation would be given along with a tree structure to be substituted for the current non-terminal.

With three syntaxes, the syntax designer has enough flexibility to try the kinds of experiments that experience has shown to be necessary. He can rearrange menu options and he can specify that a single selection will create a tree of arbitrary complexity. New problems arise,

however, because with this proposal, it is no longer true that contraction and deletion are the inverse of text creation. A holophrast might represent a non-terminal in the middle of a tree specified by the menu syntax. It is possible that this non-terminal has never been seen before by the user. In other respects as well there is no longer a direct, tangible relationship between the user's actions and the created text. The extent to which this added complexity confuses or disturbs users can only be determined by further experimentation.

6. Observations and Conclusions

Preceding chapters have shown how a system can be designed around the principle of text construction by selection of syntax rules. Each chapter has discussed the ramifications of individual portions of the system, but the system must be considered as a whole to judge the effectiveness of the concepts and of this particular implementation of those concepts. This chapter will discuss what has been learned by implementing and using Emily. A final section will consider possible extensions of this work.

The Hierarchical Hypothesis and System Implementation

Although plans for the Emily project did not include a full test of the hierarchical hypothesis, it was possible to employ programming techniques dictated by that hypothesis and observe the results. One concern was to test whether a system could be written while adhering to a hierarchical coding discipline. This discipline dictates that the system be written with few labels and be modularly divided into hierarchical levels of subroutines. Code without labels is ideal if the program is written with the aid of Emily. All the code subordinate to an IF or DO can be represented by a single holophrast so the reader can observe just the flow of control statements. When he wishes to read subordinate code, he simply expands the corresponding holophrast.

Though it was not written with the aid of Emily, the Emily implementation follows the labelless discipline fairly well. In six thousand

statements there are only forty labels, excluding labels necessary to simulate the CASE statement. Often, a label in Emily signifies code that was changed after it was first written. The label served to change the flow of control without having to rewrite a large section. The CASE statement is a generalized IF statement wherein an arithmetic expression selects one of a group of statements to be executed. Hierarchical coding requires a CASE statement. The Emily system, for example, would have used the CASE statement to decode the concrete syntax. In the present PL/I implementation, a label array must be used and this requires two labels for each case in order to avoid error messages and a subroutine call for the GOTO.

Implementation of a modular, hierarchical subroutine structure presented few difficulties. To allow separate compilation, subroutines are not nested within each other, but there are several logical levels of subroutines as shown in Figure 6.1. At the highest level are the control modules including several 'user oriented routines.' Each of these controls the sequence of operations for one or more of the buttons on the function keyboard. Just below the control level are routines to convert user requests into operations on the Emily data structures, and these routines call on a third level to convert from data structure operations into basic input/output operations. Specific requests for system operations are made by the fourth level routines. As it stands, this diagram violates the hierarchical hypothesis, for example four higher level routines refer to 'Wait for a User Action.' However, the blocks represent subroutines and the flow of control at execution time does exhibit hierarchical

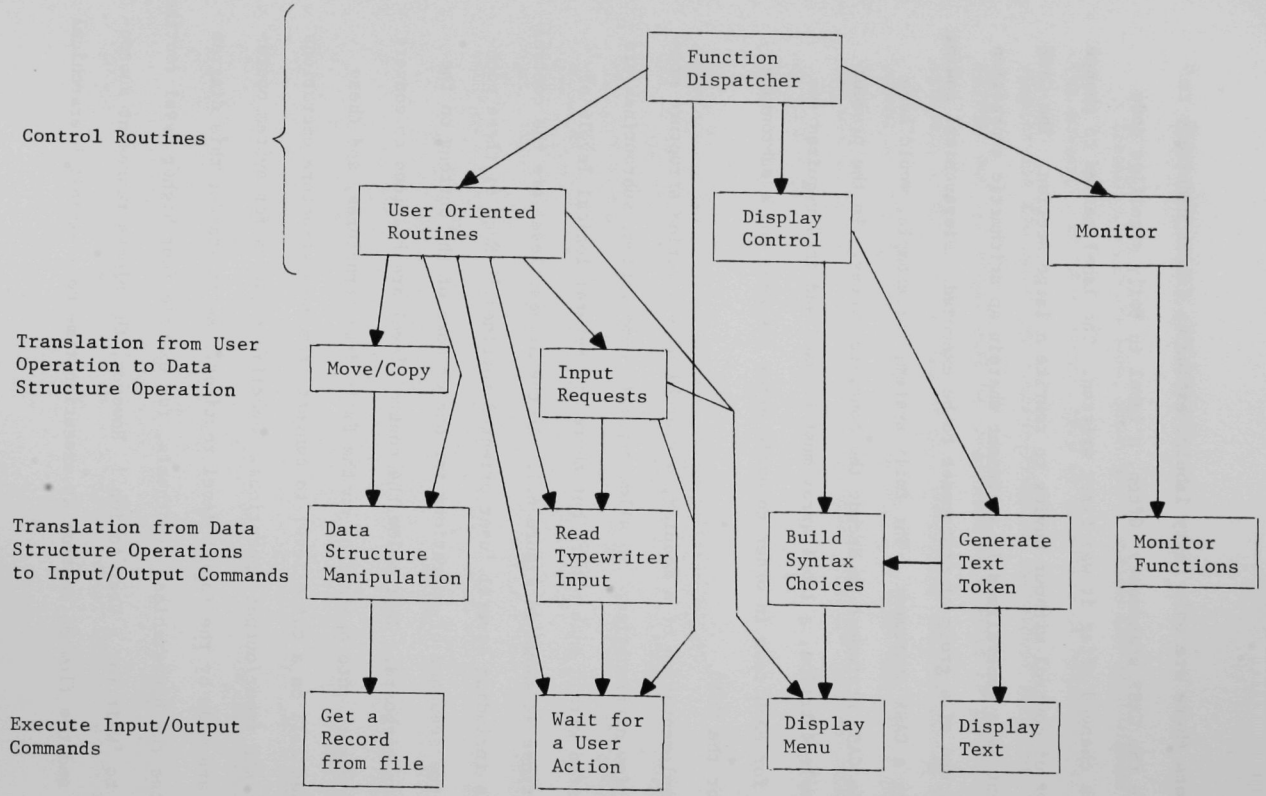


Figure 6.1. Hierarchical Structure of the Emily System

behavior. In fact, because all lines lead downward, only one sequence of lines can be active at any given time.

A hierarchical solution was also chosen for the system implementation problem of how to store the text. One method--considered in the design of an earlier system--was to store the text as a string and parse it when necessary. This approach would facilitate display generation, but offers formidable problems for syntax controlled text modification and holophrastic representation. The text would somehow have to include non-terminal symbols and the requirement for a parser would probably restrict the class of permissible languages. These problems were avoided by storing Emily text in the hierarchical structure outlined in the second chapter.

In a tree structure, each node can represent many characters so it might seem that it could be a more economical representation than strings of characters. Perhaps this is the case with a language like COBOL that has long keywords, but it is not the case with PL/I. The Emily tree for a typical PL/I routine requires roughly five or six times as much storage as the corresponding string of characters. Between twenty-five and thirty-five percent of this total is occupied by data structures for identifiers, including chains of references and block structure information. In fact, each individual reference to an identifier requires sixteen bytes, which is usually larger than the identifier itself.

When the author began Emily, he had never used a storage management system like that in PL/I where the user has explicit responsibility for

freeing allocated storage. He was curious to compare such a system with the automatic garbage collection in systems like LISP. However, most of the Emily data structure consists of nodes that are only referenced from one other node in the structure. When that reference is severed (because the user modifies text) the referenced node can be freed. For these cases, explicit freeing was a simple operation and only a small part of the procedure to sever a reference. All identifiers are linked in a symbol table and can be referenced even if they are not referenced by any text node. For this reason, identifiers are never deleted. In short, the PL/I storage management mechanisms were very convenient for this particular application.

A problem was noted with the OS/360 storage management for the space required by the various Emily modules (scatter loaded in order to use Large Core Storage). OS allocates space in multiplies of 2048 bytes. Since most modules are not multiples of any reasonable number, there was some waste space in almost all storage allocated to subroutines. This waste was usually above 12% and went as high as 20%. Fortunately, Large Core Storage is infrequently used at Argonne so the waste of space was not significant.

User Experience

Over the past year, the author has used the Emily system for a total of some 54 console hours. Though the majority of this time was spent debugging various system features, at least 14 hours were devoted to creating various versions of the PL/I syntax definition (Appendix C). Another 4 or 5 hours were spent demonstrating the system to interested observers. From this console experience it is possible to make a variety of observations concerning the system and the concepts it implements.

Because Emily is intended for programming in a higher level language, this analysis will concentrate on the console session during which I created the procedure FIND_ID listed in Appendix D. This procedure is similar to a portion of the identifier handling scheme in Emily and represents an algorithm slightly more complex than can be conveniently memorized. During the session when I created FIND_ID, I did not refer to the program listing, but reconstructed the algorithm by thinking it through. A few days later, for comparison, I used the RESCUE text editor (Joseph, 1969; Dritz, 1969) to reconstruct FIND_ID again. For the latter experiment, the console was an IBM 2741 terminal (a selectric typewriter). Statistics for both runs are summarized in Figure 6.2. It should be noted that since the experimenter served as his own subject, this analysis lacks objectivity. The major purpose in presenting these figures is to demonstrate the relative level of user activity required to use Emily. Some suggestions for more objective experiments are included in the last section of this chapter.

	Emily	RESCUE
Interactions[1]	798	115
Text interactions	459 (58%)	72 (63%)
Command interactions	339 (42%)	43 (37%)
Characters Typed	293	~1900
Lines of Text	79	67
Console Time	62 min[2]	40 min[3]
Seconds/Interaction	4.1	17.2
Corrections[4]		
Backspace	10	22
Change Recent Lines	1	8
Insert Declaration	3	2
Change Older Text	4	2
Bugs in Final Algorithm	2	3

Figure 6.2. Comparison of Emily with a Linear Text Editor

The text constructed was the procedure in Appendix D. The Emily version requires 1742 characters and the RESCUE version roughly the same.

Notes:
1. An interaction is an interrupt to the computer. The user must wait for a response before making further entries.
2. Includes 7 min. 40 sec. of data structure sketching that was remembered and not repeated for RESCUE.
3. Includes 7 min. 20 sec. to list and read text.
4. Number of occasions (each might require more than one interaction).

For the present discussion, an 'interaction' is a user action that demands a response from the computer. The user cannot proceed until he receives this response. Text interactions were those directly involved in text construction while command interactions changed the view or the text. The higher percentage of command interactions with Emily reflects two facts. First, more correction was required because the Emily version was constructed first and I had not looked at the algorithm for several months. Second, to view text with RESCUE it was only necessary to examine the typewriter output, so no explicit viewing interactions were required.

With Emily, text is constructed almost entirely by interactions, while the typewriter editor required only one interaction for each line. Few characters need be typed with Emily; of the 293 listed, over half were the text inside comments. With RESCUE all characters must be typed resulting in an elevenfold increase in the number of characters that must be typed correctly. Once text is entered, it must be displayed. Emily automatically formats the text, which takes a burden off the user, but Emily does not format the text as compactly as possible, as shown by the fact that it requires twelve extra lines to print the text. Given the shorter line length of the graphic display (39 for large characters), however, even more line breaks are required when the text is displayed. (The Emily text printing routine allows a maximum of 72 characters on a line, but the line may be indented up to half the width of the page. This feature had no affect in FIND_ID, but did cause a few line overflows while printing the PL/I syntax in Appendix C.)

Ultimately, a tool must be judged in relation to its conservation of the fundamental resource--human time. The evidence from the comparison of Emily and RESCUE is ambivalent. If one adds to the RESCUE time the seven minutes of think time spent in Emily, RESCUE still took only three-quarters as long as Emily. A number of factors mitigate this result. First, it is necessary to distinguish between the concepts employed in Emily and the current implementation of those concepts. Even within this implementation, it is possible to reorganize the text display routine to save at least a fifth of a second on every interaction. The light pen can be replaced by some device more appropriate for menu selection. These steps would reduce the interaction time with little change in the system. The only way to speed RESCUE interaction, however, is to teach the user how to type faster. (I type fairly well, so I doubt the RESCUE time can be reduced very much.)

A second mitigating factor is the cost of mistakes and the fact that, subjectively, Emily seems to reduce mistakes. With RESCUE I had three times as many short-term corrections (backspace and change recent lines) as I did with Emily. There were fewer long-term corrections in RESCUE, but these surfaced later when comparison showed three errors in the algorithm. These were not syntax errors, but were omitted statements that could only have been detected by careful testing. For example, the test for a null input string was omitted. My experience with RESCUE was that it required greater concentration merely to get the syntax correct and this concentration interferred with my analysis of the algorithm. This experience bears out the hierarchical hypothesis

that operations similar to mental processes are conducive to creative work.

Greater appreciation of the Emily facilities can be gained by careful analysis of the use of those facilities during a console session. Figure 6.3 shows the breakdown of the 339 command interactions required to generate FIND_ID with Emily.

Two-fifths of these interactions served only to locate required views of the text. Though at 4.1 seconds per interaction this amounts to 10 minutes, RESCUE required 7 2/3 minutes to list the text and, prior to that, several more minutes to examine typewriter output. Two steps can be taken to reduce viewing interactions in Emily. First, the interactions to set expansion depth to a large number ('+7') can probably be eliminated by the strategy outlined in Chapter 4. Second, a single operation can be introduced to view the tail of a list. Currently, the user must go IN to an element and then OUT one level to see the remainder of the list. Even with this strategy, it would still be preferable to retain redundant DO END pairs like the section of FIND_ID entitled '/* CHECK TO SEE IF ITS THERE */.' These pairs delimit the extent of such comments and serve to emphasize the unitary steps of the algorithm. The introduction of the extra level of nesting is wholly in keeping with the hierarchical hypothesis.

A majority of the text modification operations for FIND_ID occurred in two contexts: 'backspaces' and the IF statements to calculate CHAINNØ. In Emily, a backspace is a sequence of interactions to reject a syntax rule selection. Four interactions are required with the

Operation	How Often Used	Additional Interactions to Select Operands	
View Adjustment			
IN	12	12	
OUT	17	17	
+1 (set expansion	10	--	
+7 parameter)	25	--	
CNTRCT	12	12	
expand	29	--	
			146
Text Modification			
MOVE	3	6	
COPY	7	14	
MORE (more menu choices)	12	--	
DELEL (delete list element)	2	2	
INAFT (insert list element)	15	16	
DELETE	25	25	
'BACKSPACE' (CNTRCT last created node)	16	17	
CNTRCT	7	8	
			175
System Request			
MONITOR	3	12	
NORMAL	3	--	
			18
			339

Figure 6.3. Distribution of Emily Command Interactions

These statistics are derived from careful analysis of the console session that created Appendix D. ('BACKSPACE' includes only instances of CNTRCT necessary for backspacing as defined in the text. The corresponding deletions are included in the count for DELETE.)

present system: push CNTRCT button, light pen last generated node to contract it to a holophrast, push DELETE, and light pen the holophrast. This sequence of operations could easily be invoked by a single button. As a limited form of 'rescind last operation,' it would save the user considerable effort. (I did not realize how much effort until I analyzed the console session and discovered that I had 'backspaced' ten times. This illustrates the danger that the user of a system can, through constant use, inure himself to system inadequacies.)

Many of the DELETE's and most of the MOVE's and COPY's were involved in generating the statements to calculate CHAINNO. The method used was similar to that shown in Figure 3.1, though not as neat and straightforward. Analysis shows that this method saved at least a minute over brute force creation of the IF's. More could have been saved if the plan of Figure 3.1 had been followed.

The identifier facility has unfortunate repercussions for text modification. This facility requires that whenever a subtree of text is moved it must be scanned and all identifier reference links updated. This scan requires an inordinate amount of time and seriously degrades the system from its behavior before the identifier 'improvements' were made. Since text is moved more frequently than it is searched for identifiers, the system ought to be modified to remove most of the identifier information currently maintained. The interactive cross reference facility could still be provided, but it would require a search while the user waited. If this approach is adopted, an old problem will reappear: too many identifiers in the menu and no distinction as to whether they have been declared or not. Possibly this

problem can be attacked by maintaining block structure information and keeping a count of how often an identifier is referenced in a given block. The system could be designed so that inaccurate counts were no more than an inconvenience.

List modification operations are an important component of the Emily text modification facilities. Indeed, the operations of inserting and deleting list elements were used more frequently than MOVE and COPY. While constructing FIND_ID, every time text was changed at a location other than the current last statement, an element had to be inserted in a list of statements or declarations. The utility of the list operations was not fully realized until they had been implemented, but they turned out to be easy to use and well suited to the system style of interaction. When the list facilities were designed there was some concern over the ambiguous insertion problem (two or more lists might have elements inserted at the same point in the text). In practice, this worry did not materialize; the PL/I language is so designed that there is seldom any list ambiguity. Most lists have explicit separators, so pointing at a comma or semicolon uniquely identifies the list in which insertion is to be made.

There are still questions concerning the list feature. One is whether the menu options for list replacement provide enough protection. Lists can be terminated either by selection from the left side of the menu or by choosing the option 'EMPTY.' It may be too easy for a user to select from the left when he really intends to continue the

list by selecting from the right. During creation of FIND_ID thirty-six lists were terminated, eighteen by each method. Three of the terminations by selection from the left side of the menu were premature and the list non-terminal had to be reinserted with INAFT. (I would have terminated a greater proportion of lists by selection from the left except that the syntax for factored declarations includes list non-terminals that had to be replaced by explicit substitution of EMPTY.)

Selecting a location in a list also presents problems. To insert an element in a list, the user must point to the character immediately before the desired insertion point. Twice while creating FIND_ID I pointed at a character not followed by a list insertion point. For example, to insert in a list of statements I pointed at the character just before the semicolon rather than the semicolon itself.

To date, the major use of Emily has been the construction of syntax definitions and outlines. This thesis was first outlined with Emily and several talks have been planned in a similar manner. Generation of a PL/I syntax definition was necessary before any PL/I routines, including FIND_ID, could be built. Both definitions and outlines have an entirely hierarchical structure; both are organized as lists of elements. It should not have been surprising then that the hierarchy viewing and list operations were especially helpful for text manipulation. In fact for these simple languages, Emily behaves in a manner similar to the hierarchical editor developed by D. Engelbart (1968). Only with the PL/I syntax are the many advantages of Emily fully utilized.

Other Advantages and Disadvantages of Emily

A few experiences with Emily were not illustrated by the FIND_ID experiment. For example, two disadvantages to the Emily concept were apparent before the system was constructed. They did not appear to be major problems and, in fact, were not. The first is that text must always be manipulated in terms of its structure. This has ordinarily been advantageous, but might be a disadvantage if a user wished to treat text with a structure unlike its own. For instance it is not possible to modify spacing and carriage returns by hand. Nor is it possible to include a PL/I program fragment as part of an English text; though this facility could be added by allowing certain classes of non-terminals to be replaced by fragment names. A second problem is not as easy to solve but is, to date, only a problem with the PL/I language. The macro facility in that language may specify that certain identifiers are to be replaced by character strings before the program is compiled. These strings can include punctuation, operator symbols, and even semi-colons. But such replaceable identifiers would usually not be allowed and cannot be generated by Emily.

In the design stage, one advantage Emily seemed to have was automatic adjustment of the 'field of view,' that is, the portion of the text that occupies the display. Possibilities for automatic adjustment occur when the user expands holophrasts and later contracts them. As he expands, the top of the display might be made to descend into the portion of the text where the expansion occurred. As he contracts again, the top of the display might move up the tree. This automatic view adjustment

feature was not implemented because it seemed difficult to define and appeared to violate the principles of predictable behavior and display inertia. The result might be frequent user disorientation as he lost the thread of his text.

Disorientation in fact occurs in the one context where Emily does automatically adjust the display. When the interactive display facility is requested to display the next instance of an identifier, it finds the instance and displays it by ascending three levels in the tree. (The number of levels can be set by the user at the console.) In practice, this feature frequently leaves the user staring at a disembodied identifier instance with no immediately apparent relation to the rest of the text. This problem might be circumvented by implementing a mode in which holophrasts were specially highlighted if they met some criteria. Although the criteria 'contains non-terminals' can easily be implemented with the current system, it is difficult to see how to implement 'contains a given identifier' in such a way as to generate the display fast enough.

When the Emily system was planned, the author and his advisor expected the system to be useful for both the novice and the expert programmer though for different reasons. The novice would be guided in the construction of his program because the system would present choices. He would not have to worry about syntax errors and could concentrate on the structure of his algorithm. The expert might be aided because he could work very rapidly and introduce several characters into the finished text with each interaction. Novices who

have tried the system have had little difficulty becoming accustomed to menu selection rather than text entry.

There are several problems for the expert. First is the unsuitability of the light pen as discussed in Chapter 4. Second, there is an annoying extra level of interaction required to enter an identifier into the text. When the current non-terminal is, say <ARITHX> (that is, <arithmetic expression>), there is no difficulty in selecting an arithmetic operator. With the Emily syntactic formalism, all arithmetic operations are direct rules for <ARITHX> and the system can generate proper parentheses automatically. However, to replace <ARITHX> with an identifier, the user must first select the rule giving the identifier non-terminal, <ARITH>. Only then can he enter the character string for the identifier or select an existing identifier (from either the menu or the text).

The expert has another problem if he tries to copy an arithmetic expression from notes or a textbook. There is a consistent tendency to enter operations from left to right rather than parsing the expression to determine its structure. One of the possible solutions was suggested in Chapter 5: the syntax for arithmetic expressions can be a sequence of operands separated by operators. Another solution is to specify in the syntax that

<ARITHX> IS A CONSTANT

so the user always types in the entire arithmetic expression. A third solution would be to allow keyboard entry for any non-terminal. The

system would then have to parse the entered text if it were to be manipulated with the structure operations.

That the first two solutions can be implemented without modification to the current system demonstrates the flexibility of the Emily syntactic formalism and syntax processor. This flexibility leads to the use of Emily for an unexpected class of user, the language designer. He can design a language without facing syntactic restrictions and can immediately create programs in that language. Without the effort of learning the syntactic details of the language he can work through a number of examples to see how the language will look and feel to a user.

Future Work with the Emily Concept

With the current cost of computer time and display devices, it is difficult to justify Emily on an economic basis: but since Emily points at a new form of information and a closer relation between man and machine, and since computer costs are descending while human costs are rising, it is appropriate to consider how the Emily concepts would fit in a general interactive computer access system.

In such a system, it would be possible for an Emily-like system to be the primary interface between the user and the system. Compilers would not have to parse text and output routines could leave information in a tree structure rather than reducing it to character strings. The syntax processor is an example of the possibilities of the Emily data structure for compilation. Because the processor accesses the file directly, it avoids the need to parse the text and avoids any

question of ambiguity. In addition to program text, an entire data base might well be stored as an Emily tree. If high level titles were well chosen, the text could easily be read and required information extracted without excessive scanning. Statistical output and financial statements are other examples of hierarchical information that could be examined with Emily. For the programmer who uses a language without labels, a trace and dump facility can be implemented that saves status information in hierarchical structures (Dijkstra 1968b).

Better hardware may enhance an Emily oriented system. A critical factor in the basic Emily interactive loop is the time required to display the next set of productions. Moreover, a small number of production sets generally account for a majority of the displays. Consideration must be given to storing the most used production sets at the display rather than transmitting them each time they are needed. This storage could be accomplished with core memory at the display, but a more inventive solution is to combine a CRT and a computer controlled microfilm display, either sharing one screen or mounted side by side. The microfilm would display invariant information while the CRT would be the creative surface. Going further, a small computer controlling the displays could handle the basic Emily text creation cycle. The main computer would only manipulate structures and perform meta-functions.

As the system changes and technology improves, it will become possible to locate consoles in the user's home as well as in his office. The Emily system is based on the assumption that it is the user's primary

means of access to his data. This assumption can only be justified if the user can access his data whenever he wants. For the author, at least, this means nights and weekends as well as during 'office hours.' A serious limitation to home consoles is that a large amount of data must be displayed very rapidly. Current telephone lines can handle only a few hundred characters per second while instantaneous display modification requires at least tens of thousands. This capability may soon become available with switchable cable television networks and the telephone circuits for Picturephone® transmission.

In addition to investigation of advanced hardware and further exploration of software possibilities, the design of more effective man-computer interfaces demands increased study of how people interact with machines. Among the questions that must be answered are these:

What factors affect the success of any given feature in a given system?

Exactly how fast can users transmit and receive requests?

How much information can a user retain in his mind?

What is the most convenient way for a user to request a particular view of text?

Another question was that asked by D. Engelbart (1971): How can systems be designed to match the specifications of specific classes of users?

The next research goal of the Emily project is to use the system as a tool to help answer such questions. A user community can be encouraged to use the system by adding interactive compilation and execution. As they use the system, data can be gathered and stored

for later analysis. This analysis is somewhat easier than analysis of typewriter text because Emily text manipulation follows the structure of the text. In particular the level of detail at which the user is studying text can be determined by analysis of the holophrasts in the display. One hypothesis that can be studied is the supposition that success with Emily is correlated to the user's spatial relations ability. If true, this hypothesis points to one categorization of users that must be taken into account in system design. Through such studies the Emily system, while itself demonstrating a new user tool, can contribute to the development of systems that will further narrow the gap between man and machine.

7. Summary

The Emily text manipulation system allows a user to construct text by selecting syntax rules rather than by typing text. One reason this might be a good approach is suggested by the 'hierarchical hypothesis' as outlined in Chapter 1. This hypothesis states that (at least some) humans think in terms of hierarchical structures and that systems suited to this mode of thought are more conducive to creative work than systems supporting only strings of text. Hierarchical thinking appears in many programming contexts: programming without labels, procedure modularity, tree data structures, outlines for documentation, syntactic description. The basic Emily system was described in Chapter 2 and the many additional features of the complete system were described in Chapter 3. The advantages and disadvantages of the system are summarized in Figure 7.1.

Any interactive system must be designed so as to maximize its utility to the users. The principles behind this design in the Emily system are called user engineering principles, as outlined in Chapter 4. Two principles were basic to the design of Emily. The first of these, 'selection not entry', was followed in the basic Emily text construction cycle and several other contexts in the system: file selection, identifier and name selection, sub-operation selection, and even selection of the level to which the display should ascend. The other principle, 'predictable behavior', was employed consistently throughout the system to make its behavior understandable in terms of a small set of concepts that the user can perceive with a little practice.

Apriori Advantages

User cannot make simple syntax errors. (5)
System can manipulate text in any syntax, even if ambiguous. (5)
Neophytes are guided while building text. (6)
Experts require fewer keystrokes than when typing text. (6)
User does not enter format characters (spaces and carriage returns). (6)
Display automatically adjusts to a change in view of text. (6)
Text display replaces flowcharts. (6)
User has flexible control of 'field of view'. (6)
System can provide an interactive cross reference facility based on the block structure. (6)
User can modify text in terms of its structure, rather than its character string representation. (6)
Programs can be written to modify text in terms of its structure. (3)

Unexpected Advantages

Valuable tool for language designers. (6)
Very flexible facilities for viewing text at different levels of detail. (6)
List modification is especially helpful. (6)

Apriori Disadvantages

Text can only be manipulated in terms of its structure. (6)
Emily is inconsistent with the PL/I macro facility. (6)

Unexpected Disadvantages

Light pen is poor for selecting choices. (4)
Identifiers and fragment names migrate in the menu. (4)
Emily syntactic formalism will not permit modification of the menu without modifying the text structure. (5)
For PL/I, it is difficult to write a declaration syntax that will properly restrict identifier classes. (5)
Attempts to provide shorthand menu selections lead to too many choices. (5)
Extra menu selection usually required before user can enter an identifier. (6)
Arithmetic expressions are difficult to copy from handwritten notes. (6)
Identifier facility requires too much storage and slows down more useful system features. (6)

Figure 7.1. Advantages and Disadvantages of Emily

Each comment is amplified further in the chapter noted in parentheses.

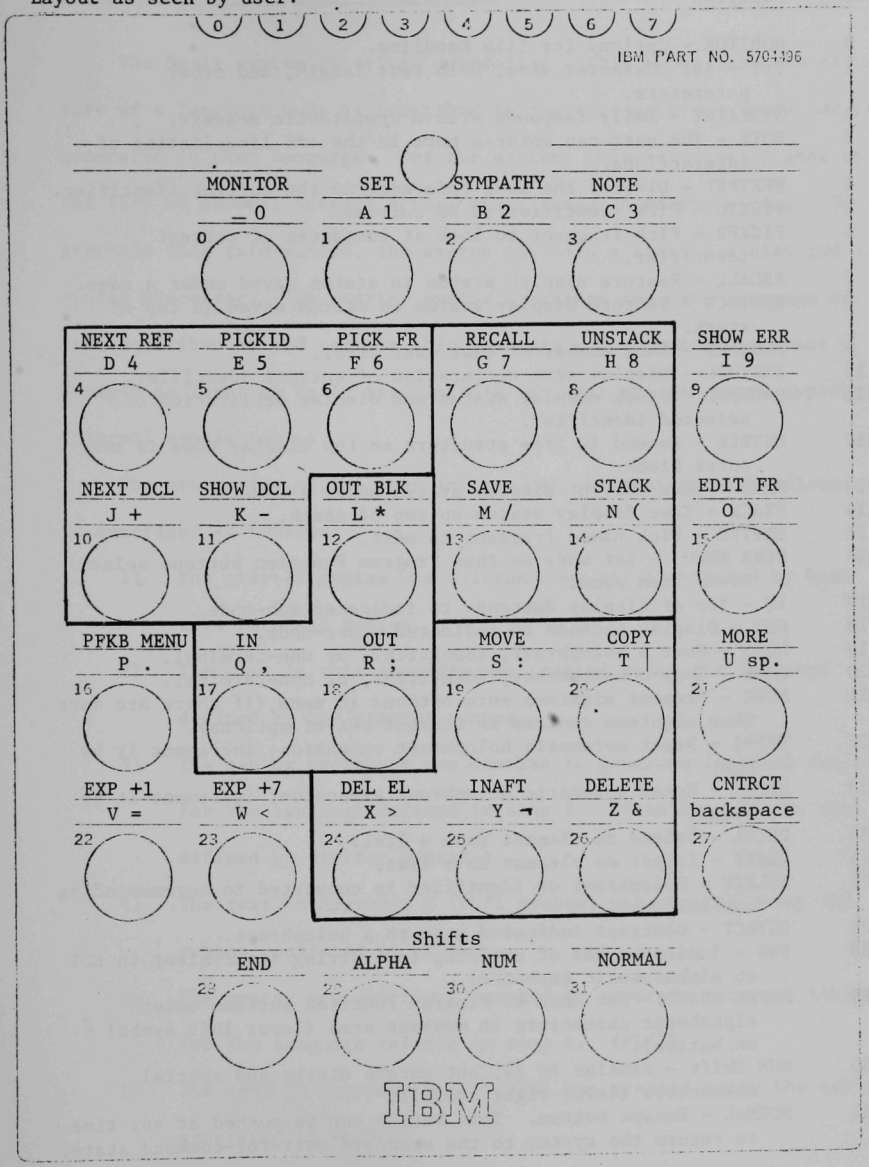
Complete understanding of the capabilities of Emily requires detailed knowledge of the formalism employed to describe the syntax of a programming language. This formalism has been described in Chapter 5 and contains three features not found in ordinary BNF-like formalisms. Identifiers and block structure are described in the formalism so the system can keep track of all references to identifiers. Indentation and carriage returns are specified in the syntax so the system can automatically format the display. Finally, to provide automatic parenthesization of arithmetic expressions and to provide more flexible display formatting, the syntax permits conditional display operations. Conditions that can be tested include the contents of any subnode of a node and the identity of the father of the node.

The experience gained from implementing and using Emily is related in the sixth chapter. The design of the system followed the hierarchical hypothesis in procedure modularization, design of the data structure, and avoidance of labels. An experiment comparing Emily with a typewriter based text editor showed that construction of a simple text took somewhat longer with Emily, but the user made fewer mistakes. The mistakes when using the typewriter seemed to stem from the greater effort at concentration required to ensure correct syntax. Chapter 6 closed with several possibilities for extension of this work. Emily can be particularly instrumental in statistical analysis to determine the characteristics of users.

I read my sentence steadily,
Reviewed it with my eyes,
To see that I made no mistake
In its extremest clause...

Emily Dickinson

Layout as seen by user:



Button	Function
0	MONITOR - Options for file handling.
1	SET - Set character size, holo text length, and other parameters.
2	SYMPATHY - Emily responds with a sympathetic message.
3	NOTE - The user can enter a note in the off line listing of interactions.
4	NEXTREF - Display the next reference to the current identifier.
5	PICKID - Pick identifier to be current.
6	PICKFR - Pick fragment to look at instances of current identifier.
7	RECALL - Restore display status to status saved under a name.
8	UNSTACK - Restore display status to status saved on top of stack.
9	SHOWERR - Show number of last user error.
10	NEXTDCL - Display next declaration of current identifier.
11	SHOWDCL - Stack display status and display declaration of selected identifier.
12	OUTBLK - Ascend in tree structure so top display node is next outer block.
13	SAVE - Save current display status under a name.
14	STACK - Save display status on top of stack.
15	EDITFR - Pick named fragment to edit.
16	PFKB MENU - Set mode so that Program Function buttons select items from menu.
17	IN - Top of display descends to indicated sub-node.
18	OUT - Display ascends to indicated super-node.
19	MOVE - Move a holophrast, identifier, or non-terminal.
20	COPY - Copy a holophrast, identifier, or non-terminal.
21	MORE - Present eighteen more options in menu (if there are more than eighteen choices in current set of options).
22	EXP+1 - Reset automatic holophrast expansion; increment it by one.
23	EXP+7 - Reset automatic holophrast expansion; increment it by seven.
24	DELEL - Delete an element from a list.
25	INAFT - Insert an element in a list.
26	DELETE - Holophrast or identifier is converted to corresponding non-terminal.
27	CNTRCT - Contract indicated node to a holophrast.
28	END - Indicates end of entering text string (equivalent to EOT on alphanumeric keyboard).
29	ALPHA Shift - Set mode so Program Function buttons enter alphabetic characters in message area (lower left symbol on button).
30	NUM Shift - Similar to 29, but enters digits and special characters (lower right symbol).
31	NORMAL - Escape button. This button can be pushed at any time to return the system to the standard wait-for-command state.

Appendix B. Emily Syntactic Formalism

The Emily system is syntax controlled. This means that the structure of a language must be described to the system before texts can be generated in that language. For the system, this description must be in the form of several integer arrays and a large character string. To generate this information, the system includes a starter syntax and a syntax processor. The starter syntax describes the same language as that described by the text in this appendix. The syntax processor accepts as input an Emily text in this language and generates appropriate internal syntax tables.

The generation of the text in this and the next two appendices can be described as follows:

1) The starter syntax was written down and translated by hand into internal tables.
2) The text in this Appendix was created using the language defined by the starter syntax.
3) The syntax processor was invoked to generate internal tables for the language defined in step 2. (Same language as that defined by starter syntax.)
4) The text in Appendix C (PL/I syntax) was created using the tables generated in step 3.
5) The syntax processor was invoked to generate internal tables for the language defined in step 4. (PL/I)
6) The text in Appendix D (FIND_ID) was created using the tables generated in step 5.

As a syntax, the text in this appendix has fifty-five syntax rules and twenty non-terminal symbols. The internal representation of this text occupies 10243 bytes with 2452 of these required for 78 references to 23 distinct identifiers. If the text were stored as a character string with no redundant blanks, it would occupy only 1623 bytes, but if it were stored as card images, it would be 7280 bytes.

```
SYNTAX FOR SYNTAXES

<SYNDEF> IS OUTER N-T

<SYNDEF> : <TITLE> NL? NL '<' <NT> '> IS OUTER N-T' NL? NL <PR/*/NL?>
        NL? <PRINTSPEC/*?/NL?>

PRODUCTIONS

   <PR> : '<' <NT/DECL> '>' INDENT+6 <RHS/*/NL?>
        : '<' <NT/DECL> '> IS A CONSTANT'
        : NL <TITLE> NL? INDENT+3 NL <PR/*/NL?> NL? <PRINTSPEC/*?/NL?>
        : '<' <NT/DECL> '> IS AN ID'
   <RHS> : ' : ' INDENT+6 <ITEM/*/' '>
        : ' : BLOCK ' INDENT+6 <ITEM/*/' '>
        : ' : (' <LABEL/DECL> ') ' INDENT+6 <ITEM/*/' '>
        : ' : (' <LABEL/DECL> ') BLOCK ' INDENT+6 <ITEM/*/' '>
   <PRINTSPEC> : '<' <NT> '*>' INDENT+6 <RHS>
   <TITLE> IS AN ID
```

ITEMS

FIRST EIGHT

```
<ITEM> : '<' <NT> '>'
       : '<' <NT> '/' <STAR> '>'
       : '<' <NT> '/DECL>'
       : '<' <NT> '/DECL/' <STAR> '>'
       : ''' <STRING> '''
       : 'INDENT' <INDENT>
       : 'NL?'
       : 'XIT'
```

SECOND EIGHT

```
<ITEM> : 'IFT (' <ITEM/*/' '> ')'
       : 'IFF (' <ITEM/*/' '> ')'
       : 'SND' <SNDNO> '=(' <LABEL> ')'
       : 'SND' <SNDNO> ' IN (' <LABEL> '-' <LABEL> ')'
       : 'DAD=(' <LABEL> ')'
       : 'IFTXIT'
       : 'IFT()'
       : 'NOT'
```

THIRD EIGHT

```
<ITEM> : 'LIST'
       : 'NL'
       : 'SND' <SNDNO>
       : 'DPTH' <DEPTH>
       : 'TAB'
       : 'TABS (' <TAB/*?/', '> ')'
       : 'MVRT' <INDENT>
       : 'COL' <COLNO>
```

```
FOURTH SEVEN

   <ITEM> : 'SNDNT' <SNDNO>
          : 'SNDHCLC' <SNDNO>
          : 'SNDNT' <SNDNO>
          : 'DPTH>' <DEPTH>
          : 'CCL<' <CCLNO>
          : 'TRUE'
          : 'FALSE'

<STAR> : '*?'
       : '*'
       : '*?/'' <STRING> '''
       : '*/'' <STRING> '''
       : '*?/' <ITEM/*/' '>
       : '*/' <ITEM/*/' '>
       : '*?/(' <LABEL> ')'
       : '*/(' <LABEL> ')'
       : '?'


SPECIAL NON-TERMINALS

   <NT> IS AN ID
   <LABEL> IS AN ID
   <STRING> IS A CONSTANT
   <INDENT> IS A CONSTANT
   <DEPTH> IS A CONSTANT
   <SNDNO> IS A CONSTANT
   <COLNO> IS A CONSTANT
   <TAB> IS A CONSTANT
```

Appendix C. Emily Syntax for PL/I

To aid in understanding the organization of the PL/I syntax, the following is a view of the syntax that can be displayed on the 2250 screen. In this view, the length of holophrasts is set to sixteen so the titles are not cut too short.

```
!PR:PROCEDURE SYNTAX!
!PR:STATEMENT SYNTAX!

DECLARATIONS

   !PR:DATA DECLARATION!
   !PR:INPUT OUTPUT DEC!
   !PR:PROGRAM CONTROL !

EXPRESSION SYNTAX

   !PR:GENERAL <ARG> : !
   !PR:ARITHMETIC EXPRE!
   !PR:BIT EXPRESSIONS !
   !PR:CHARACTER EXPRES!
   !PR:POINTER EXPRESSI!

!PR:IDTYPES <AREA> !
```

This PL/I syntax is the largest text that has so far been generated with Emily. The internal representation occupies 65825 bytes of which 14596 are required for 598 references to 100 identifiers. If the text were stored as a string, it would require 11351 characters, exclusive of redundant blanks. The printed text occupies 556 lines, so it would require 44480 bytes if stored in card image form.

As an (abbreviated) description of PL/I, this syntax has 336 rules and 102 non-terminal symbols. Of the latter, eleven are identifiers, twenty-one are constants, and the rest have rules specifying replacements.

PL/I SYNTAX

<PROC> IS OUTER N-T

PROCEDURE SYNTAX

```
   <PROC> : <ENTRYNM/DECL> ': PROC' INDENT+4 <PROC BODY> INDENT+0 NL? '
               END ' SND1 ';'
          : <ENTRYNM/DECL> <ENTRYNM/DECL/*?/(MULTDCL)> ': PROC' INDENT+4 <
               PROC BODY> INDENT+0 NL? 'END ' SND1 ';'
   <PROC BODY> : BLOCK SNDMT1 IFF (' ') <PROC OPT/*?/' '> ';' NL? '/* ' <
               COMMENT> ' */' NL? <STMT/*/NL?>
   <PROC OPT> : '(' <PARM/*?/', '> ')'
          : <DATA ATTR>
          : 'RECURSIVE'
          : 'OPTIONS (MAIN)'
          : 'OPTIONS (TASK)'
          : 'OPTIONS (MAIN, TASK)'
   <PARM> : <ARITH>
          : <BIT>
          : <CHAR>
          : <PTR>
          : <STRUCT>
          : <ENTRYNM>
          : <AREA>
          : <EVENT>
          : <LABEL>
          : <TASK>
          : <FILENM>
   <ENTRYNM> IS AN ID
   <COMMENT> IS A CONSTANT
   <ENTRYNM*> : (MULTDCL) ': ' <ENTRYNM> LIST <ENTRYNM/DECL/*?/(MULTDCL)>
```

STATEMENT SYNTAX

```
<STMT> : CCL20 '/* ' <COMMENT> ' */'
       : NL '/* ' <COMMENT> ' */' NL NL
       : <ASGN STMT>
       : 'DECLARE ' INDENT+4 NL? <DATA DCL/*/',' NL?> ';'
       : NL <PROC> NL
       : 'CALL ' INDENT+4 <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '> ';'
       : (DOEND) 'DO;' INDENT+4 NL? <STMT/*/NL?> INDENT+0 NL? 'END;'
       : 'DO WHILE (' INDENT+8 <BITX> ');' INDENT+4 NL? <STMT/*/NL?> INDENT+0
           NL? 'END;'
       : 'DO ' INDENT+8 <VAR> ' = ' <DO SPEC> ';' INDENT+4 NL? <STMT/*/NL?>
           INDENT+0 NL? 'END;'
       : 'DO ' INDENT+8 <ARITHV> ' = ' <ARITHX> ' TO ' <ARITHX> ';' INDENT+4
           NL? <STMT/*/NL?> INDENT+0 NL? 'END;'
       : (IFTHEN) 'IF ' INDENT+8 <BITX> ' THEN ' SND2=(DOEND) IFF (INDENT+4
           NL?) IFT (INDENT+0) <STMT>
       : 'IF ' INDENT+8 <BITX> ' THEN ' SND2=(DOEND) IFF (INDENT+4 NL?) IFT (
           INDENT+0) <STMT> SND2=(IFTHEN) IFT (NL? 'ELSE;') INDENT+0 NL? 'ELSE ' <
           STMT>
       : 'RETURN;'
       : 'RETURN (' INDENT+4 <EXPR> ');'
       : 'ALLOCATE ' INDENT+4 <ALLOC ITEM/*/', '> ';'
       : 'FREE ' INDENT+4 <FREE ITEM/*/', '> ';'
       : <I/O STMT>
       : BLOCK 'ON ' <CONDITION> ' BEGIN;' INDENT+4 NL? <STMT/*/NL?> INDENT+0
           NL? 'END; /* ON UNIT */'
       : ';'
       : 'EXIT;'
       : 'DECLARE ' INDENT+4 <FILE DCL/*/', '> ';'
       : 'DECLARE ' INDENT+4 <ENTRY DCL/*/', '> ';'
       : MVRT-3 <LABEL/DECL> ':'
       : 'GO TO ' <LABEL> ';'
```

```
<DO SPEC> : <EXPR>
          : <EXPR> ' TO ' <EXPR>
          : <EXPR> ' BY ' <EXPR>
          : <EXPR> ' TO ' <EXPR> ' BY ' <EXPR>
          : <EXPR> ' WHILE(' <BITX> ')'
          : <EXPR> ' TO ' <EXPR> ' WHILE(' <BITX> ')'
          : <EXPR> ' BY ' <EXPR> ' WHILE(' <BITX> ')'
          : <EXPR> ' TO ' <EXPR> ' BY ' <EXPR> ' WHILE(' <BITX> ')'
          : <DO SPEC> <DO SPEC/*/', '>
<ASGN STMT> : <ARITH> ' = ' <ARITHX> ';'
          : <ARITHV> ' = ' <ARITHX> ';'
          : <VAR> ' = ' <EXPR> ';'
          : <VAR/*/', '> ' = ' <EXPR> ';'
          : <BITV> ' = ' <BITX> ';'
          : <CHARV> ' = ' <CHARX> ';'
          : <PTRV> ' = ' <PTRX> ';'

STATEMENT OPTIONS

   <ALLOC ITEM> : <STRUCT>
                : <STRUCT> ' IN (' <AREA> ')'
                : <STRUCT> ' SET (' <PTR> ')'
                : <STRUCT> ' IN (' <AREA> ') SET (' <PTR> ')'
   <FREE ITEM> : <STRUCT>
                : <PTR> '->' <STRUCT>
                : <STRUCT> ' IN (' <AREA> ')'
                : <PTR> '->' <STRUCT> ' IN (' <AREA> ')'
```

```
I/O STATEMENTS

   <I/O STMT> : 'GET ' INCENT+4 <GET OPT/*/' '> ';'
          : 'PUT ' INDENT+4 <PUT OPT/*/' '> ';'
          : <OTHER I/O>
   <GET OPT> : 'FILE (' <FILENM> ')'
          : 'STRING (' <CHAR> ')'
          : 'LIST (' <GET VAR/*/', '> ')'
          : 'DATA (' <GET VAR/*/', '> ')'
          : 'EDIT (' <GET VAR/*/', '> ') (' <FORMAT ITEM/*/', '> ')'
          : 'SKIP'
          : 'COPY'
   <PUT OPT> : 'FILE (' <FILENM> ')'
          : 'STRING (' <CHAR> ')'
          : 'LIST (' <PUT EXPR/*/', '> ')'
          : 'DATA (' <PUT EXPR/*/', '> ')'
          : 'EDIT (' <PUT EXPR/*/', '> ') (' <FORMAT ITEM/*/', '> ')'
          : 'SKIP'
          : 'COPY'
   <OTHER I/O> IS A CONSTANT
   <FORMAT ITEM> IS A CONSTANT
   <GET VAR> IS A CONSTANT
   <PUT EXPR> IS A CONSTANT
<CONDITION> IS A CONSTANT
```

DECLARATIONS

DATA DECLARATIONS

```
<DATA DCL> : <ARITH/DECL> SNDMT2 IFF (' ') <ARITH ATTR/*?/' '>
       : <BIT/DECL> SNDMT2 IFF (' ') <BIT ATTR/*?/' '>
       : <CHAR/DECL> SNDMT2 IFF (' ') <CHAR ATTR/*?/' '>
       : <PTR/DECL> SNDMT2 IFF (' ') <PTR ATTR/*?/' '>
       : <AREA/DECL> ' AREA (' <SIZE> ')' SNDMT3 IFF (' ') <STORCL/?>
       : <CTL DCL>
       : '(' <DATA DCL/*/', '> ')' SNDMT2 IFF (' ') <DATA ATTR/*?/' '>
       : NL? '1 ' <STRUCT/DECL> SNDMT2 IFF (' ') INDENT+4 <STORCL/?> ',' NL?
            <STRUCT DCL/*/',' NL?>
<STRUCT DCL> : <LEVEL> ' ' <DATA DCL>
       : <LEVEL> ' ' <STRUCT/DECL> ',' INDENT+4 NL? <STRUCT DCL/*/',' NL?>
       : <LEVEL> ' ' <STRUCT/DECL> 'ALIGNED' ',' INDENT+4 NL? <STRUCT DCL/*/'
            ',' NL?>
       : <LEVEL> ' ' <STRUCT/DECL> 'UNALIGNED' ',' INDENT+4 NL? <STRUCT DCL/
            */',' NL?>
       : <LEVEL> ' ' <STRUCT/DECL> ' LIKE ' <STRUCT>
<ARITH ATTR> : 'FIXED BIN'
       : 'FLOAT BIN'
       : 'FIXED DEC'
       : 'FLOAT DEC'
       : '(' <PRECISON> ')'
       : '(' <PRECISON> ', ' <SCALE> ')'
       : 'REAL'
       : 'COMPLEX'
       : 'PICTURE "' <PICTURE> '"'
       : '(' <BOUNDS/*/', '> ')'
       : <STORCL>
       : 'INITIAL (' <NUMBER/*/', '> ')'
       : 'INITIAL CALL ' <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '>
       : 'ALIGNED'
       : 'UNALIGNED'
       : 'DEFINED ' <PARM> SNDMT2 IFF (' ') NOT IFT() <DEF SUBS/*?/', '>
            SNDMT3 IFTXIT ' POSITION (' <POSITION/?> ')'
```

```
<BIT ATTR> : 'BIT (' <LENGTH> ')'
       : 'VAR'
       : '(' <BOUNDS/*/', '> ')'
       : <STORCL>
       : 'INITIAL (' <BIT STRING/*/(BITS)> ')'
       : 'INITIAL CALL ' <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '>
       : 'ALIGNED'
       : 'UNALIGNED'
       : 'DEFINED ' <PARM> SNDMT2 IFF (' ') NOT IFT() <DEF SUBS/*?/', '>
            SNDMT3 IFTXIT ' POSITION (' <POSITION/?> ')'
<CHAR ATTR> : 'CHAR (' <LENGTH> ')'
       : 'VAR'
       : '(' <BOUNDS/*/', '> ')'
       : <STORCL>
       : 'INITIAL (' <CHAR STR/*/(CHARS)> ')'
       : 'INITIAL CALL ' <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '>
       : 'ALIGNED'
       : 'UNALIGNED'
       : 'DEFINED ' <PARM> SNDMT2 IFF (' ') NOT IFT() <DEF SUBS/*?/', '>
            SNDMT3 IFTXIT ' POSITION (' <POSITION/?> ')'
<PTR ATTR> : 'PTR'
       : 'OFFSET (' <AREA> ')'
       : '(' <BOUNDS/*/', '> ')'
       : <STORCL>
       : 'INITIAL (' <CONSTANT/*/', '> ')'
       : 'INITIAL CALL ' <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '>
       : 'ALIGNED'
       : 'UNALIGNED'
       : 'DEFINED ' <PARM> SNDMT2 IFF (' ') NOT IFT() <DEF SUBS/*?/', '>
            SNDMT3 IFTXIT ' POSITION (' <POSITION/?> ')'
```

```
<DATA ATTR> : 'FIXED BIN'
       : 'FLOAT BIN'
       : 'FIXED DEC'
       : 'FLOAT DEC'
       : '(' <PRECISON> ')'
       : '(' <PRECISON> ', ' <SCALE> ')'
       : 'REAL'
       : 'COMPLEX'
       : 'PICTURE '' <PICTURE> '''
       : 'BIT (' <LENGTH> ')'
       : 'CHAR (' <LENGTH> ')'
       : 'VAR'
       : 'PTR'
       : 'OFFSET (' <AREA> ')'
       : '(' <BOUNDS/*/', '> ')'
       : <STORCL>
       : 'INITIAL (' <CONSTANT/*/', '> ')'
       : 'INITIAL CALL ' <ENTRYNM> SNDMT2 NOT IFT() <ARG/*?/', '>
       : 'ALIGNED'
       : 'UNALIGNED'
       : 'DEFINED ' <PARM> SNDMT2 IFF (' ') NOT IFT() <DEF SUBS/*?/', '>
          SNDMT3 IFTXIT ' POSITION (' <POSITION/?> ')'
<STORCL> : 'AUTO'
       : 'BASED (' <PTR> ')'
       : 'STATIC INT'
       : 'STATIC EXT'
       : 'CTL INT'
       : 'CTL EXT'
<BOUNDS> : <HBOUND>
       : <LBOUND> ':' <HBOUND>
```

```
   ID TYPES

      <CONSTANT> IS A CONSTANT
      <HBOUND> IS A CONSTANT
      <LBOUND> IS A CONSTANT
      <LENGTH> IS A CONSTANT
      <LEVEL> IS A CONSTANT
      <PICTURE> IS A CONSTANT
      <POSITION> IS A CONSTANT
      <PRECISON> IS A CONSTANT
      <SCALE> IS A CONSTANT
      <SIZE> IS A CONSTANT
      <DEF SUBS> IS A CONSTANT
   <BIT STRING*> : (BITS) ''' <BIT STRING> ''B' LIST ', ' <BIT STRING/*/(
                   BITS)>
   <CHAR STR*> : (CHARS) ''' <CHAR STR> ''' LIST ', ' <CHAR STR/*/(CHARS)>

INPUT OUTPUT DECLARATIONS

   <FILE DCL> : <FILENM/DECL> ' FILE' <FILE ATTR/*?/' '>
   <FILE ATTR> IS A CONSTANT

PROGRAM CONTROL DCL'S

   <ENTRY DCL> : <ENTRYNM/DECL> ' ENTRY'
   <CTL DCL> : <LABEL/DECL> ' LABEL' SNDMT2 NOT IFT() <LABEL/*/', '>
          : <EVENT/DECL> ' EVENT'
          : <TASK/DECL> ' TASK'
```

EXPRESSION SYNTAX

GENERAL

```
<ARG> : <ARITHX>
      : <BITX>
      : <CHARX>
      : <AREA>
      : <ENTRYNM>
      : <EVENT>
      : <FILENM>
      : <LABEL>
      : <PTRX>
      : <TASK>
      : <STRUCT>
<EXPR> : <ARITHX>
       : <BITX>
       : <CHARX>
       : <PTRX>
       : <VAR>
<VAR> : <AREA>
      : <ARITHV>
      : <BITV>
      : <CHARV>
      : <EVENT>
      : <LABEL>
      : <PTRV>
      : <STRUCT>
      : <TASK>
```

ARITHMETIC EXPRESSIONS

```
<ARITHX> : <ARITH>
         : <ARITHV>
         : <NUMBER>
         : '(' <ARITHX> ')'
         : '+' SND1 IN (ADD-POW) IFT() <ARITHX>
         : '-' SND1 IN (ADD-POW) IFT() <ARITHX>
         : (BIT) '(' <BITX> ')'
         : (CHAR) '(' <CHARX> ')'
         : (ADD) <ARITHX> ' + ' <ARITHX>
         : (SUB) <ARITHX> ' - ' SND2=(ADD) IFT() <ARITHX>
         : (MUL) SND1 IN (ADD-SUB) IFT() <ARITHX> '*' SND2 IN (ADD-SUB) IFT() <
             ARITHX>
         : (DIV) SND1 IN (ADD-SUB) IFT() <ARITHX> '/' SND2 IN (ADD-MUL) IFT() <
             ARITHX>
         : (POW) SND1 IN (ADD-DIV) IFT() <ARITHX> '**' SND2 IN (ADD-DIV) IFT()
             <ARITHX>
<ARITHV> : <ARITH>
         : <ARITH> ' (' <ARITHX/*/', '> ')'
         : <STRUCT/*/'.'> '.' <ARITH>
         : <STRUCT/*/'.'> '.' <ARITH> ' (' <ARITHX/*/', '> ')'
         : <ENTRYNM>
         : <ENTRYNM> ' (' <ARG/*/', '> ')'
         : <PTR> '->' <ARITH>
         : <STRUCT/*/'.'> '.' <PTR> '->' <ARITH>
         : <PTR> '->' <STRUCT/*/'.'> '.' <ARITH>
         : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <ARITH>
         : <PTR> '->' <ARITH> ' (' <ARITHX/*/', '> ')'
         : <STRUCT/*/'.'> '.' <PTR> '->' <ARITH> ' (' <ARITHX/*/', '> ')'
         : <PTR> '->' <STRUCT/*/'.'> '.' <ARITH> ' (' <ARITHX/*/', '> ')'
         : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <ARITH> ' (' <
             ARITHX/*/', '> ')'
```

BIT EXPRESSIONS

```
<BITX> : <BIT>
       : <BITV>
       : ''' <BIT STRING> ''B'
       : '(' <BITX> ')'
       : '(' <ARITHX> ')'
       : '(' <CHARX> ')'
       : (REL) <ARITHX> <RELOP> <ARITHX>
       : SND1 IN (AND-OR) IFT() <BITX> <RELOP> SND2 IN (AND-OR) IFT() <BITX>
       : <CHARX> <RELOP> <CHARX>
       : <PTRX> ' = ' <PTRX>
       : <PTRX> ' ¬= ' <PTRX>
       : (NOT) ' ¬' SND1 IN (REL-OR) IFT() <BITX>
       : (AND) SND1=(OR) IFT() <BITX> ' & ' SND2=(OR) IFT() <BITX>
       : (OR) <BITX> ' | ' <BITX>
<BITV> : <BIT>
       : <BIT> ' (' <ARITHX/*/', '> ')'
       : <STRUCT/*/'.'> '.' <BIT>
       : <STRUCT/*/'.'> '.' <BIT> ' (' <ARITHX/*/', '> ')'
       : <ENTRYNM>
       : <ENTRYNM> ' (' <ARG/*/', '> ')'
       : <PTR> '->' <BIT>
       : <STRUCT/*/'.'> '.' <PTR> '->' <BIT>
       : <PTR> '->' <STRUCT/*/'.'> '.' <BIT>
       : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <BIT>
       : <PTR> '->' <BIT> ' (' <ARITHX/*/', '> ')'
       : <STRUCT/*/'.'> '.' <PTR> '->' <BIT> ' (' <ARITHX/*/', '> ')'
       : <PTR> '->' <STRUCT/*/'.'> '.' <BIT> ' (' <ARITHX/*/', '> ')'
       : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <BIT> ' (' <ARITHX/
           */', '> ')'
```

```
   <RELOP> : ' = '
           : ' ¬= '
           : ' < '
           : ' ¬< '
           : ' > '
           : ' ¬> '
           : ' >= '
           : ' <= '
   <BIT STRING> IS A CONSTANT

CHARACTER EXPRESSIONS

   <CHARX> : <CHAR>
           : <CHARV>
           : ''' <CHAR STR> '''
           : <CHARX> '||' <CHARX>
           : '(' <ARITHX> ')'
           : '(' <BITX> ')'
           : 'SUBSTR(' <CHARX> ', ' <ARITHX> ', ' <ARITHX> ')'
           : 'INDEX(' <CHARX> ', ' <CHARX> ')'
   <CHARV> : <CHAR>
           : <CHAR> ' (' <ARITHX/*/', '> ')'
           : <STRUCT/*/'.'> '.' <CHAR>
           : <STRUCT/*/'.'> '.' <CHAR> ' (' <ARITHX/*/', '> ')'
           : <ENTRYNM>
           : <ENTRYNM> ' (' <ARG/*/', '> ')'
           : <PTR> '->' <CHAR>
           : <STRUCT/*/'.'> '.' <PTR> '->' <CHAR>
           : <PTR> '->' <STRUCT/*/'.'> '.' <CHAR>
           : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <CHAR>
           : <PTR> '->' <CHAR> ' (' <ARITHX/*/', '> ')'
           : <STRUCT/*/'.'> '.' <PTR> '->' <CHAR> ' (' <ARITHX/*/', '> ')'
           : <PTR> '->' <STRUCT/*/'.'> '.' <CHAR> ' (' <ARITHX/*/', '> ')'
           : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <CHAR> ' (' <ARITHX
               /*/', '> ')'
   <CHAR STR> IS A CONSTANT
```

```
   POINTER EXPRESSIONS

      <PTRX> : <PTR>
             : <PTRV>
             : 'NULL'
      <PTRV> : <PTR>
             : <PTR> ' (' <ARITHX/*/', '> ')'
             : <STRUCT/*/'.'> '.' <PTR>
             : <STRUCT/*/'.'> '.' <PTR> ' (' <ARITHX/*/', '> ')'
             : <ENTRYNM>
             : <ENTRYNM> ' (' <ARG/*/', '> ')'
             : <PTR> '->' <PTR>
             : <STRUCT/*/'.'> '.' <PTR> '->' <PTR>
             : <PTR> '->' <STRUCT/*/'.'> '.' <PTR>
             : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <PTR>
             : <PTR> '->' <PTR> ' (' <ARITHX/*/', '> ')'
             : <STRUCT/*/'.'> '.' <PTR> '->' <PTR> ' (' <ARITHX/*/', '> ')'
             : <PTR> '->' <STRUCT/*/'.'> '.' <PTR> ' (' <ARITHX/*/', '> ')'
             : <STRUCT/*/'.'> '.' <PTR> '->' <STRUCT/*/'.'> '.' <PTR> ' (' <ARITHX/
                 */', '> ')'

   ID TYPES

      <AREA> IS AN ID
      <ARITH> IS AN ID
      <BIT> IS AN ID
      <CHAR> IS AN ID
      <EVENT> IS AN ID
      <FILENM> IS AN ID
      <LABEL> IS AN ID
      <NUMBER> IS A CONSTANT
      <PTR> IS AN ID
      <STRUCT> IS AN ID
      <TASK> IS AN ID
```

Appendix D. PL/I Program Created with Emily

The procedure FIND_ID listed in this appendix represents the algorithm used for storing and locating identifiers within the Emily system. The pointer SYMBOLS points at an ordered sequence of IDND's, chained on the IDNEXT field. Each element of the array CHAINHD contains a pointer to the pointer at the first identifier with some given initial letter.

The text given contains the two errors generated during the console session; both are in the section 'CHECK TO SEE IF ITS THERE'. The greater-than should be a less-than, and the next two statements should be surrounded by 'ELSE DO;' and 'END'.

The tree representation of FIND_ID requires 9072 bytes, including 3037 bytes for 112 references to 24 identifiers. Without redundant blanks, this text would occupy 1742 bytes as a character string. If stored on cards, it would occupy 79 cards or 6320 bytes.

```
FIND_ID: PROC (ID) PTR;
    /* LOCATE ID IN SYMBOL TABLE */
                    /* RETURN PTR TO IT */
    DECLARE
        ID CHAR (32) VAR;

    /* DEFINE SYMBOL TABLE */

    DECLARE
        SYMBOLS PTR STATIC EXT,
        CHAINHD (27) PTR STATIC INT,
        1 IDNODE BASED (IDP),
            2 (IDTYPE, IDLEN) FIXED BIN,
            2 (IDNEXT, IDATTR) PTR,
            2 IDCHARS CHAR (NREF REFER (IDLEN)),
        IDP PTR,
        NREF FIXED BIN,
        FIRST_TIME BIT (1) STATIC INT INITIAL ('1'B);
    DECLARE
        (PREVPTR, CURRPTR) PTR,
        PTRPTR PTR BASED (PREVPTR),
        CHAINNO FIXED BIN,
        TC CHAR (1);
```

```
IF FIRST_TIME THEN DO;
    SYMBOLS = NULL;
    DO CHAINNO = 1 TO 27;
            /* INIT TO PTR AT CHAIN */
        CHAINHD (CHAINNO) = ADDR (SYMBOLS);
    END;
    FIRST_TIME = '0'B;
END;
IF LENGTH (ID) = 0 THEN
    RETURN (NULL);
DO;
            /* FIND PROPER CHAIN */
    TC = SUBSTR(ID, 1, 1);
    IF TC < 'A' THEN
        CHAINNO = 1;
    ELSE IF TC ¬> 'I' THEN
        CHAINNO = 2 + UNSPEC (TC) - UNSPEC ('A');
    ELSE IF TC < 'J' THEN
        CHAINNO = 1;
    ELSE IF TC ¬> 'R' THEN
        CHAINNO = 11 + UNSPEC (TC) - UNSPEC ('J');
    ELSE IF TC < 'S' THEN
        CHAINNO = 1;
    ELSE IF TC ¬> 'Z' THEN
        CHAINNO = 20 + UNSPEC (TC) - UNSPEC ('S');
    ELSE CHAINNO = 1;
END;
```

```
    DO;
                    /* CHECK TO SEE IF ITS THERE */
        PREVPTR = CHAINHD (CHAINNO);
        CURRPTR = PREVPTR->PTRPTR;
        DO WHILE (CURRPTR ¬= NULL);
            IF LENGTH (ID) = CURRPTR->IDLEN THEN
                IF ID = CURRPTR->IDCHARS THEN
                    RETURN (CURRPTR);
            IF ID > CURRPTR->IDCHARS THEN
                CURRPTR = NULL;
            PREVPTR = ADDR (CURRPTR->IDNEXT);
            CURRPTR = PREVPTR->PTRPTR;
        END;
    END;

    /* MAKE A NEW IDENTIFIER NODE */

    NREF = LENGTH (ID);
    ALLOCATE IDNODE SET (CURRPTR);
    CURRPTR->IDTYPE = IDTYPE#;
    CURRPTR->IDCHARS = ID;
    CURRPTR->IDATTR = NULL;
    CURRPTR->IDNEXT = PREVPTR->PTRPTR;
    PREVPTR->PTRPTR = CURRPTR;
    DO CHAINNO = CHAINNO + 1 TO 27 WHILE(CHAINHD (CHAINNO) = PREVPTR);
        CHAINHD (CHAINNO) = ADDR (CURRPTR->IDNEXT);
    END;
    RETURN (CURRPTR);
END FIND_ID;
```

References

The bracketed numbers indicate the page where the paper is referenced.

(Backus, 1959) Backus, J. W., 'The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.' *Proc. International Conf. on Information Processing*, UNESCO, 1959, pp. 125-132. [1, 15]

(Böhm and Jacopini, 1966) Böhm, C. and G. Jacopini, 'Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.' *Comm. ACM 9* (May, 1966), pp. 366-371. [7]

(Bratman, 1968) Bratman, Harvey, Hiram G. Martin, and Ellen C. Perstein, 'Program Composition and Editing with an On-line Display.' *AFIPS Proc. V. 33 pt. 2* (FJCC), 1968, pp. 1349-1360. [8]

(Bush, 1945) Bush, Vannevar, 'As We May Think.' *The Atlantic Monthly*, (July, 1945), pp. 101-108. [8]

(Carmody, 1968) Carmody, Steven, Theodor H. Nelson, David Rice, and Andries Van Dam, *A Hypertext Editing System for the /360*. Brown University, October, 1968 (unpublished). [9]

(Cameron, 1967) Cameron, Scott H., Duncan Ewing, and Michael Liveright, 'DIALOG: A Conversational Programming System with a Graphical Orientation.' *Comm. ACM V. 10*, 6 (June, 1967), pp. 349-357. [8]

(Dijkstra, 1968a) Dijkstra, E. W., 'GO TO Statement Considered Harmful.' (Letter to the editor) *Comm. ACM 11*, 3 (March, 1968), pp. 147-148. [7]

(Dijkstra, 1968b) Dijkstra, E. W., 'A Reply by E. W. Dijkstra.' (Reply to letter by J. R. Rice) *Comm. ACM 11*, 8 (Aug., 1968), pp. 538 and 541. [7, 94]

(Dritz, 1969) Dritz, K., 'User's Guide to the PL/I Text Editor and Syntax Processor.' Applied Math. Div., Argonne National Laboratory, Argonne, Illinois, April, 1969 (unpublished). [81]

(Engelbart, 1968) Engelbart, Douglass C., and William K. English, 'A Research Center for Augmenting Human Intellect.' *AFIPS Proc. V. 33 pt. 1* (FJCC), 1968, pp. 395-410. [9, 89]

(Engelbart, 1971) Engelbart, D. C., Stanford Research Institute, Menlo Park, Calif. private communication. [42, 95]

(George, 1967) George, J. E., *The SPIRES Scope Demonstration System.* Stanford Linear Accelerator Center, CGTM 33, Nov., 1967. [8]

(George, 1968) George, J. E., *Calgen - An Interactive Picture Calculus Generation System.* Computer Science Department, Stanford Univ., CS 114, 1968. [8, 46]

(Gladwin, 1970) Gladwin, R., 'The Utilization of Graphic Display Units as the Main Form of Computer Input.' *Computer Graphics 70 International Symposium*, April, 1970, (paper not in proceedings; author is with Kay and Co., England). [46]

(Hansen, 1969) Hansen, W. J., 'Compact List Representation: Definition, Garbage Collection, and System Implementation.' *Comm. ACM 12*, 9 (Sept., 1969), pp. 499-507. [58]

(Hansen, 1970) Hansen, W. J., 'Graphic Editing of Structured Text.' Proceedings of *Computer Graphics 1970 International Symposium*, Brunel University, Uxbridge, Middlesex, England, V. 3, Section 7, April, 1970. [*vi*]

(IBM, 1968) IBM Corporation, *Graphic Subroutine Package (GSP)*. Form No. C27-6932, Programming Publications, Kingston, New York, 1968. [13]

(IBMUK, 1968) IBM United Kingdom Laboratories, Ltd., *PL/I Reference Manual*. Form No. C28-8201, Programming Publications, Hursley Park, Winchester, Hampshire, England, 1968. [13]

(Joseph, 1969) Joseph, A. F. and R. L. Logan, 'RESCUE, A Time-Sharing System.' Applied Math. Div., Argonne Nat'l Lab., Argonne, Ill. 1969 (unpublished). [81]

(Koch, 1969) Koch, K. and F. Schwarzenberger, *A System for Syntax-Controlled Editing of Formula Text*. Presented at Newcastle Seminar on Automated Publishing Systems, Newcastle, England, Sept., 1969. [73]

(McCarthy, 1962) McCarthy, John, et al., *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Mass., 1962. [6]

(McCarthy, 1963) McCarthy, John, 'Towards a Mathematical Science of Computation.' In *Information Processing 1962* (C. M. Popplewell, Ed.) North-Holland Publ. Co., Amsterdam, 1963, pp. 21-28. [60]

(McCarthy, 1967) McCarthy, John, Dow Brian, Gary Feldman, and John Allen, 'THOR - A Display Based Time Sharing System.' *AFIPS Conf. Proc. V. 30* (SJCC), 1967, pp. 623-633. [8]

(Miller, 1968) Miller, R. B., 'Response Times in Man-Computer Conversational Transactions.' *AFIPS Conf. Proc. V. 33* (FJCC), 1968, Part 1, pp. 267-277. [42]

(Mitchell, 1970) Mitchell, James G., *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. Dept. of Comp. Sci., Carnegie Mellon University, June, 1970. [42]

(Nelson, 1965) Nelson, T. H., 'A File Structure for the Complex, the Changing, and the Indeterminate.' *Proc. ACM 20th Natl. Conf.*, 1965, pp. 84-100. [9]

(Reynolds, 1970) Reynolds, J. C., 'GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept.' *Comm. ACM 13*, 5 (May, 1970), pp. 308-319. [13]

(Rice, 1970) Rice, D. E., *Computer-Assisted Text Manipulation*. Division of Applied Mathematics, Brown University, June, 1970. [8]

(Ridsdale, 1970) Ridsdale, B., 'The Visual Display Unit for Data Collection and Retrieval.' Proceedings of *Computer Graphics 70 International Symposium*, Brunel University, Uxbridge, Middlesex, England, V. 3, Section 2, April, 1970. [46]

(Schorre, 1964) Schorre, D. V., 'META-II: A syntax oriented compiler writing language.' *Proc. ACM 19th Natl. Conf.*, 1964, p. D1.3. [68]

(Smith, 1969) Smith, L. B., *The Use of Man-Machine Interaction in Data-Fitting Problems*. SLAC Report No. 96, Stanford Linear Accelerator Center, Stanford, California, March, 1969. [42, 46]

(Stanford, 1968) Stanford University Computing Center. 'Wylbur Reference Manual.' Appendix E of *Campus Facility Users Manual*. February, 1968. [8]

(Thompson, 1968) Thompson, K., *QED Text Editor*. Bell Telephone Laboratories, Murray Hill, New Jersey, March, 1968 (unpublished). [8]

(Williams, 1970) Williams, R., 'On the Application of Graph Theory to Computer Data Structures.' Proceedings of *Computer Graphics 70 International Symposium*, Brunel University, Uxbridge, Middlesex, England, V. 2, Section 3, April, 1970. [6]

(Wirth, 1971) Wirth N., 'Program Development by Stepwise Refinement.' *Comm. ACM 14*, 4 (April, 1971), pp. 221-227. [6]